THE VMATRIX: A BACKWARD-COMPATIBLE SOLUTION FOR IMPROVING THE INTERACTIVITY, SCALABILITY, AND RELIABILITY OF INTERNET APPLICATIONS

A DISSERTATION SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND THE COMMITTEE ON GRADUATE STUDIES OF STANFORD UNIVERSITY IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Amr A. Awadallah

June 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Mendel Rosenblum)
Principal Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Hector Garcia-Molina)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David Mazières)

Approved for the University Committee on Graduate Studies.

ABSTRACT

Currently, most Internet services are pre-allocated to servers statically; which leads to sluggish interactivity, low availability, limited scalability, and delay fairness issues.

In this thesis, we present a novel architecture that solves these issues without requiring significant code changes (i.e., backward-compatible). This architecture is called the vMatrix, and it is an overlay network of virtual machine monitors (VMMs). A VMM encapsulates the state of the machine in a virtual machine file, which could then be migrated and activated on any real machine running the VMM software.

We identified three challenging Internet problems that previous solutions failed to address in a backward-compatible way. We then implemented a prototype of the vMatrix and studied it in detail as a solution for these three challenges:

Dynamic Content Distribution: Moving services closer to the Internet edge, thus reducing latency and rendering such services more interactive and available for end users.

Server Switching: Sharing a pool of servers between more than one service, thus leveraging the benefits of statistical multiplexing to reduce overall hardware requirements for running a group of distinct Internet services.

Equi-ping Game Server Placement: Placing game servers at optimized equi-ping locations to improve the fairness of multi-player first-person-shooter games by reducing the delay differential between participating players.

We also demonstrate additional side benefits, including on-demand replication for absorbing flash crowds (in case of a newsworthy event like a major catastrophe) and faster recovery times for improved overall reliability.

"*The Matrix is everywhere. It is all around us. Even now, in this very room. You can see it when you look out your window or when you turn on your television. You can feel it when you go to work... when you go to church... when you pay your taxes*", Morpheus, The Matrix (1999).

ACKNOWLEDGMENTS

I would like to thank my advisor, Mendel Rosenblum, for the guidance he has provided me through my Ph.D., as well as the members of my reading committee, Hector Garcia-Molina and David Mazières. I would like to especially thank Hector for helping during a rather hard time for me at Stanford in 1997. I would like to thank my former advisors, Nick McKeown, especially for his relentless advice on how to create effective presentations, and Fouad Tobagi for giving me the opportunity to come to Stanford. I also would like to thank Armando Fox for intermediate research that we did together to try to abstract all the Internet Service architectures out there. I would like to thank Abe Taha for his help in porting over a Yahoo search service to the vMatrix framework (Chapter 5). I want to thank Yahoo!, Inc for paying my tuition during the later stages of my studies, and for their support to experiment with some of the Yahoo services within the vMatrix framework.

I will not forget my office mates at Stanford: Youngmi, Pankaj, Pablo, Sundar, Chetan, Brad, Onil, Adisak, Ismail, Krishna, Lorenz, Wael, Athina, Oskar, Chuck, Lorie, Paul, Andy, Ben, and Da. I would also like to thank the members of the Egyptian and Islamic communities at Stanford.

Last, but not least, I would like to thank those who really made it possible for me to do my Ph.D. at Stanford because of their patience and emotional support: my wife (Shirin), my kids (Ahmed, Lina, Sarah, Minnah), my parents, my parents-in-law, my brother, and my sister. My love and gratitude goes to all of you.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

AJAX. Asynchronous Javascript And XML.

CDN. Content Distribution Network

CPA. Content Placement Agent

CSS. Cascading Style Sheets

DHTML. Dynamic HTML

DOM. Document Object Model

FPS. First Person Shooter Game

HN. Hosting Network

HTML. HyperText Markup Language

MMORPG. Massive Multi-player Online Role Playing Game

OS. Operating System

RM. Real Machine

RTS. Real-Time Strategy Game

VM. Virtual Machine

LO. Local Oracle

VMM. Virtual Machine Monitor

XML. eXtensible Markup Language

XHTML. Extensible HTML

XSLT. Extensible Style Language Transformation

Chapter 1 INTRODUCTION

1.1 Internet Services

As illustrated in Figure 1, an Internet service is an application that users access over the Internet, typically using a browser over HTTP, but can also include other mechanisms, such as a game client accessing a game server using UDP. Popular examples of HTTP Internet services (aka Web Apps) would be Hotmail, eBay, MySpace, Wikipedia, and Yahoo. Examples of non-HTTP services would be Skype, Google Earth, Counter Strike, and Halo (online multi-player games).

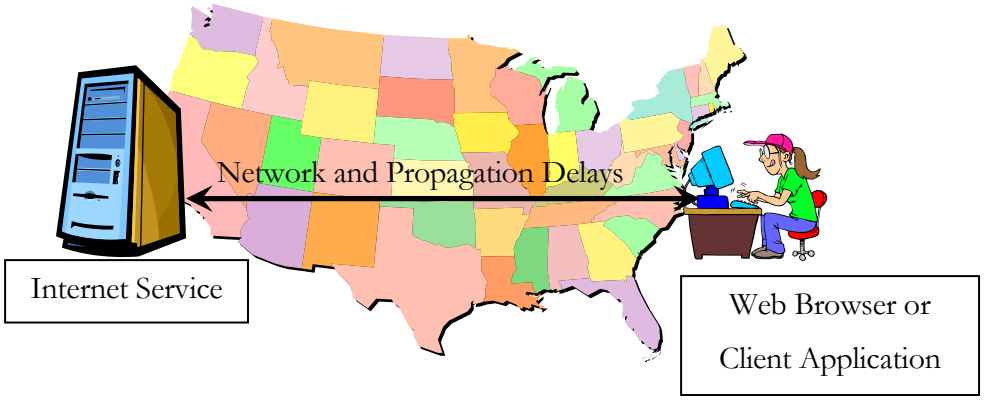

Figure 1: Example of Internet Service.

1.2 Problems Facing Internet Services

The main advantages of Internet applications over traditional desktop native applications are:

(1) They allow software publishers to update and maintain the applications easily without distributing and installing software on potentially millions of client computers,

(2) The ubiquitous access for end users allows them to use the services and associated data from home, work, school, or any place where the client application is installed, and

(3) The wide-area nature of these services allows for collaboration and peer-to-peer sharing of information.

However, the disadvantages of Internet applications, compared to desktop and LAN applications, are:

(1) Sluggish Interactivity, since the further the server is from the end-user, the longer it will take for the end user to see the reaction to their action. Although the backbone of the Internet and last-mile connectivity are getting faster every day, network delays are still very high due to ever increasing demand (especially due to the popularity of peer-to-peer video downloading services ala BitTorrent). Furthermore, network propagation delay is limited by the speed of electromagnetic signals, which is a fact we will always have to live with.

(2) Limited Scalability, since most Internet services are currently statically provisioned, as in a fixed pool of servers that is assigned to handle the load for a given service. If demand spikes for a given services demand (an example is the CNN.com meltdown on Sept 11th, 2001 [107]), it is currently very difficult to quickly repurpose servers from another service that is not using all of its resources.

(3) Limited Availability, since the farther away the user is from the server, the more likely the network in-between might get disconnected. If we are able to move the service closer to the user, then we will reduce the probability of network disconnection (since the packets will cross a smaller number of network hops).

(4) Delay Unfairness, since a number of gaming clients communicating with a centralized game server will perceive different times for the state updates, which allows some clients to see the future of other clients, thus giving them an unfair advantage.

We would like to keep the benefits of Internet applications, but, at the same time, we want the positioning and provisioning of the Internet service to be mindful of the number and position of users accessing it to avoid the disadvantages listed above.

To solve the problems of interactivity, scalability, availability, and delay fairness described above, we would like the ability to easily move services between servers to bring them closer to the end users.

Currently, service mobility is primarily possible within a few standardized application server frameworks, such as ATG Dynamo, IBM WebSphere, and BEA WebLogic [16][51][21], but, within those frameworks, library versions and operating system release mismatches can lead to inter-operability problems that hinder such mobility. In addition, it is not common for large-scale Internet services to use such application server frameworks in the first place (they are targeted more toward *intranet* apps with a few thousand users, as opposed to Internet applications with millions of users). 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 often than not, Internet service infrastructures grow in an evolutionary fashion rather than a revolutionary one.

The main difficulty with 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, most importantly, on people (developers and system administrators). Simply copying the code of the service is not possible since the target machines must have exactly the same environment for the code to run unchanged, which is not common in practice. The library versions that work with one service might cause another service to fail when run on the same server.

The classic operating system concept of virtual machine monitors (VMMs) provides us with a core building block for achieving server mobility without violating the dependencies mentioned above.

1.3 Virtual Machine Monitors

A virtual machine monitor virtualizes the real machine at the hardware layer (CPU, Memory, I/O), and exports a virtual machine (VM), which exactly mimics what a real machine would look like.

VMMs were introduced in the 1970s by IBM [50] to arbitrate access to hardware of an expensive mainframe machine between a number of client operating systems, and to provide 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 they then were resurrected in the late 90s for the x86 architecture by VMware, Inc. [104] as a byproduct of the Stanford

Disco [39] project. Their current main use is consolidation of a number of services onto a single server without having these services interfere with each other.

Our key observations are: (1) VMMs encapsulate the state of the machine in a virtual machine file, which could then be moved and activated on any real machine running the VMM software; and (2) VMMs are transparent to the service, i.e., backward-compatible, and thus, require no coding changes for the service to work on top of existing architectures.

1.4 The vMatrix*

The vMatrix is an overlay network of virtual machine monitors that enables us to dynamically move services between servers. Such a network can improve the interactivity, scalability, availability, and delay fairness of existing Internet services without the need for significant code or architecture changes (i.e., a key advantage of this framework is that it is backward-compatible for existing deployed systems).

This framework improves interactivity because it enables us to move the services closer to the end users who need them. It improves availability, since proximity implies fewer points of failure, but we can also reduce recovery time by having pre-booted, suspended VM instances, as will be discussed in later chapters. It improves scalability, since we can instantiate additional "copies" of the servers as demand dictates and easily repurpose servers from a less loaded service. Finally, it

* 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 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. We use this analogy as a tool to help our readers understand the concept and remember the name of the project.

improves delay fairness, since we can move the service into a position that is equidistant from the participating users.

The vMatrix solves the software dependency problem, since the whole service is transferred with the OS, libraries, code, modules, and code that the service depends upon. It solves the hardware dependency problem, since the VMM can lie to the overlying OS about the hardware resources available to it (e.g., memory size), thus mimicking the same hardware environment for that service regardless of the real hardware of the hosting real machine (though this might lead to performance degradation). It also solves the people dependency problem by presenting the developers and system administrators with the same isolation model that they are familiar with.

We did not attempt to build a VMM, but rather, we reference existing software for the x86 architecture from VMware, Inc. [104]. Note that similar VMM software is also available from other providers (e.g., Microsoft Virtual PC [73], or Xen Source [111]), but we choose VMware due to their close relationship with Stanford University and also because they provide a server class VMM.

The distinguishing advantages of the vMatrix framework are:

(1) Decoupling of service from server allowing for service mobility, which improves the interactivity, scalability, availability, and delay-fairness of Internet services.

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

(3) Presenting the developers and system administrators with the same machine isolation model that they are familiar with

(4) Economies of scale by leveraging the fact that this network can be shared among many different types of services rather than being fully dedicated to one service

The main disadvantage of the vMatrix framework is the overhead of virtualization. This overhead consists of two parts; the first is the CPU and I/O overhead and the second is the large size of the VM files. We do not address the former as it relates to the VMM implementation itself, but such overheads have improved significantly recently (on the order of 5% or less depending on work load). We predict that such overheads will get even better as hardware manufacturers are now integrating virtualization into their CPUs, memory chips, and I/O controllers. The latter issue of large VM file sizes can be addressed by pre-caching and differential chain coding techniques [30].

1.5 Three applications

We built a prototype of the vMatrix and used it to solve three challenging Internet problems. We then evaluated the efficacy of the vMatrix as a solution to those problems and contrasted it to other existing solutions (e.g., ghosting, application servers, packagers, and OS virtualization).

The three applications are:

Dynamic Content Distribution [9]: Moving services closer to the Internet edge, thus reducing latency and rendering such services more interactive for end users (Chapter 4)

Server Switching [10]: Sharing a pool of servers between more than one service, thus leveraging the benefits of statistical multiplexing to reduce overall system cost (Chapter 5)

Equi-ping Game Server Placement [11]: Placing game servers at optimized locations to improve the delay (ping) fairness of multi-player first-person-shooter games (Chapter 6)

1.6 Contribution

The main contributions of this thesis are two-fold:

(1) An architecture that improves the interactivity, scalability, availability, and delay-fairness of Internet services without requiring significant code changes (i.e., backward-compatible). This architecture is called the vMatrix and it is an overlay network of virtual machine monitors.

(2) We identified three challenging Internet problems (see section 1.5) that previous solutions failed to address in a backward-compatible way. We then implemented a prototype of the vMatrix and studied it in detail as a solution for those problems, while contrasting it to previous work.

1.7 Thesis Outline

In Chapter 2 we review Internet Services and Virtual Machine Monitors for the benefit of our readers who are not familiar with them. In Chapter 3 we provide an overview of the vMatrix architecture at a high level, but then provide more details about the implementation in Chapters 4, 5, and 6 as it relates to each one of the demonstrated applications. We conclude in Chapter 7 .

Chapter 2 BACKGROUND

2.1 Internet Services

As discussed in Chapter 1 , Internet Services are appealing since they allow software publishers to update applications easily without distributing the new code to millions of computers; end users can access such services from almost anywhere with the proper client, and multiple users can collaborate, share information, or play together.

However, unlike Desktop Applications, Internet Applications are remote to the user, which presents many challenges—specifically, interactivity is affected due to network delays, availability is affected due to connection loss, scalability is affected due to static provisioning, and multi-player fairness is affected due to delay unfairness.

2.1.1. A Typical Internet Service Architecture

Currently, most large-scale Internet applications are structured after the architecture presented in Figure 2. The architecture consists of these main components:

(1) A global load balancer is responsible for redirecting the traffic of a given user request based on the proximity, availability, and load conditions of the different data centers.

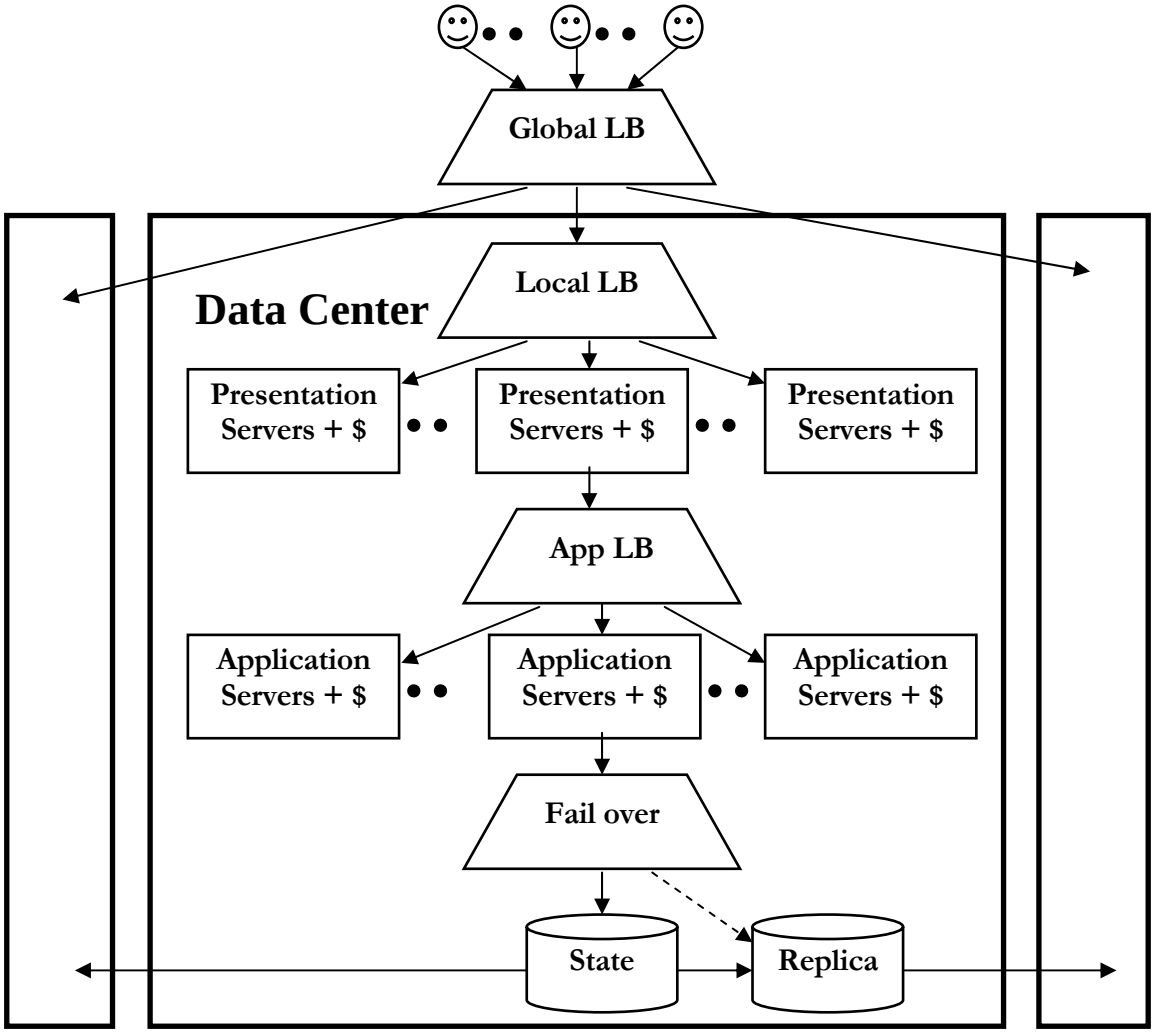

Figure 2: A Typical Internet Application Architecture
($ = cache, LB = Load Balancer).

(2) Once the request is sent to a given data center, the local load balancers distribute the traffic among a number of presentation servers (and associated cache, which we denote as $ in the figure).

(3) The presentation servers are responsible for packaging the final HTML to be served back to the user. The presentation servers send requests to the

application servers to provide the data to be included in response to the user.

(4) The application servers, in turn, obtain this data from back-end databases, which are replicated for reliability reasons. The application servers might perform some processing on the data before passing it to the presentation servers.

As we will discuss in the following chapters, it is the goal of this work to provide a backward-compatible solution that allows for the mobility of the front-end presentation servers from the main data center and closer to the end users to improve the interactivity, availability, scalability, and delay-fairness of Internet Applications.

In the following sections, we present three challenging Internet service problems, for which there are, currently, no good backward-compatible solutions.

2.1.2. Dynamic Content Distribution

Internet Content Distribution is the problem of moving content as close as possible to the end users who consume that content. This is primarily done to reduce the latency of accessing that content; however, it also has the side benefit of improving availability. There are a number of currently deployed solutions for the distribution of static content (e.g., images, PDF documents, large downloadable files, or video streams). However, the problem of Dynamic Content Distribution remains an elusive one. In this section, we present an example for Dynamic Content Distribution and discuss the issues that make this a difficult problem, particularly for multi-tier Internet services.

The benefits of distribution and replication are as follows:

(1) **Improving the response time perceived by end users**: By moving the content closer to the network edge, where the users are directly connected to their Internet service providers (ISPs), it is possible to significantly reduce the overall response time for retrieving content since packets travel over a smaller number of hops. Research that has measured the benefits of CDNs shows that they, in fact, provide a valuable service by avoiding the slowest servers [65].

(2) **Improving overall availability and up-time**: Distributing the content over a number of geographically disbursed locations provides better overall availability since, if one of the servers is down, or if the network connectivity to it is lost; then another nearby server could always be found. It also offers stronger protection against DoS (Denial of Service) attacks, because it is harder to attack all of the geographically dispersed servers. Recent simulation results [18], based on actual connectivity traces, show that distributing code inside the network can result in an order of magnitude improvement of availability.

(3) **On-Demand replication to absorb flash crowd requests**: By allowing content to be cloned and replicated on demand in different locations, flash crowds can be dynamically absorbed. A flash crowd is an unpredicted increase in web requests, such as an unforeseen surge in stock market activity might cause.

(4) **Reducing the overall bandwidth consumed in the network**: By placing the content closer to the end users, the content packets cross a smaller number of links, and thus, consume less aggregate bandwidth.

This is validated by CDN hit ratio analytic modeling [96] and by proxy cache measured results [55][38].

To illustrate the problem of Dynamic Content Distribution, consider a map service (e.g., maps.yahoo.com) where the users fill in a web form with an address, and, in return, they receive a street map. Notice that caching the returned map image is useless since while many users could be using this service at the same time, they are most likely interested in different destination addresses.

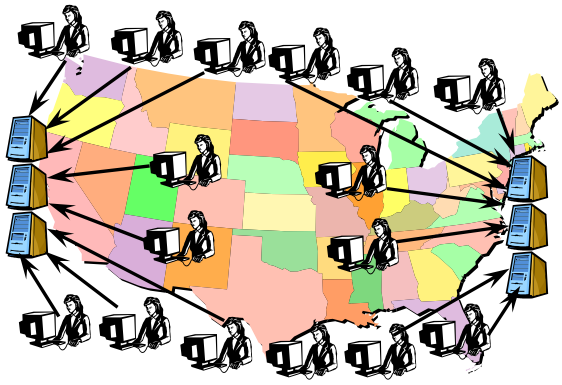

Figure 3: Single-tier architecture with static mirroring.

However, the underlying code that renders this page and the geographical database it depends upon are very static in nature. Typically, this web service is built out of a single-tier architecture, where the necessary code and geographic data are self-contained within a single server. A load-balancer is used in front of a farm of identical map servers to scale to a large number of users. In addition, the server cluster is replicated on the East and West coast primarily for reliability reasons. A typical system is illustrated in Figure 3.

It is cost prohibitive for the service providers to operate server clusters in more than a couple of data centers; indeed, this is one of the main value-added features of content –distribution networks (CDNs) for static content distribution. Static mirroring does not provide real-time distribution and replication. However, if we install the servers inside transportable virtual machines (which is no different than installing them inside real machines), and allow them to be moved in real time to demand hot spots, then we can move the servers closer to the users, as shown in Figure 4.

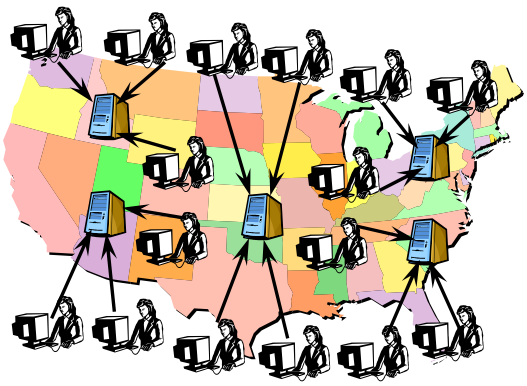

Figure 4: Single-tier architecture with mobile virtual machine servers to enable dynamic content distribution and replication.

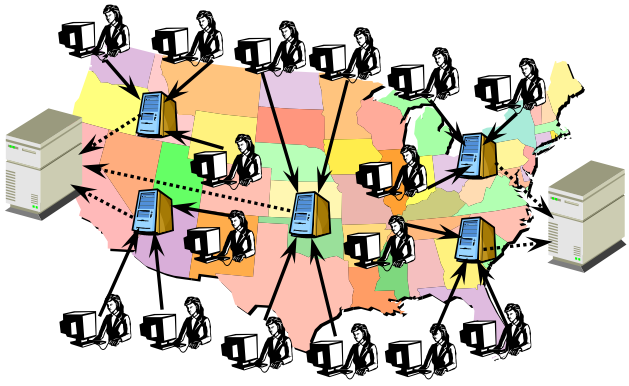

Figure 5: Two-Tier architecture with virtual private networks from the front-end servers to back-end databases.

However, the scenario is slightly more complicated if the front-end web servers need to query a back-end advertising database to display ads on the results web page, or need to access a user profile database to look up personalization features for the users. In this case, connectivity is needed between the transported front-end machines and the back-end databases, as illustrated in Figure 5. In section 4.4 we discuss in detail the trade-offs and implications of porting a two-tier architecture to the vMatrix framework.

2.1.3. Static Provisioning versus Dynamic Provisioning

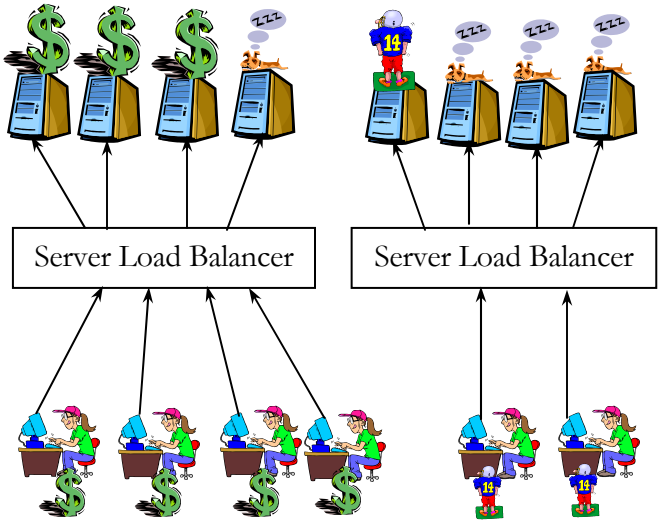

Figure 6: Statically pre-provisioned services.

The second challenge that we identified, which the vMatrix platform addresses, is how to share servers between a pool of different services. In this section, we provide an example for that problem, then tackle the solution in detail in Chapter 5 .

Yahoo! Inc. provides many popular Internet services. Consider two of them: a financial service that provides information about various stocks and mutual funds

(finance.yahoo.com), and a sports news service that provides 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 four for the financial service and four for the sports service, as illustrated in Figure 6. The load profile for the financial service is such that it is busy during weekday mornings/afternoons, and almost idle on weekday evenings/nights and weekends. In contrast, the sports service load profile is such that it is busy on weekends and weekday evenings/nights, and almost idle on weekday mornings/afternoons. Thus, if we take a snapshot of these services on the 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, resulting in a non-efficient use of the available resources (those are the servers illustrated by sleeping dogs in Figure 6). Note that it is not the case that a number of servers will be completely idle, but rather, that all servers will be operating at a portion of their full capacity.

The services are usually statically separated in this manner because it is 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 others.

(2) **People dependencies**: the developers and system administrators responsible for one service would not like to deal with the consequences of actions performed 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 about memory size, hard-disk space, or other hardware resources that

might be violated when another service shares with it the same server, or when the service code is moved to a server that does not satisfy these requirements.

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 inefficient use of the available resources. Note that this inefficiency is not only 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).

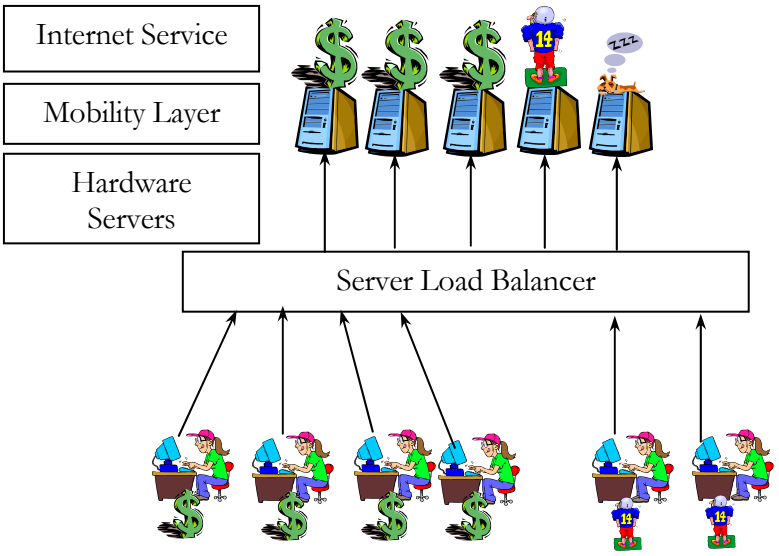

Figure 7: Server Switching (Dynamic Allocation).

Another disadvantage is the lack of 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 few hours to a few

days until enough additional servers are re-allocated from other services to the surging service, which is usually too late.

If we can dynamically move services between servers, then we can share the Real-Machines (RMs) between both services. This leads to efficient resource utilization and a reduction in the total number of required RMs. Looking at the same snapshot in Figure 6, which required 8 RMs, we now only need 4 RMs, as is illustrated in Figure 7; however, a fifth 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. In Chapter 5 , we discuss the details of how the vMatrix provides a solution for sharing servers between services.

2.1.4. Multi-player FPS Game Servers

Another popular Internet application is First-Person-Shooter (FPS) multi-player games. We observed that this application can also benefit from a backward-compatible solution, enabling server mobility.

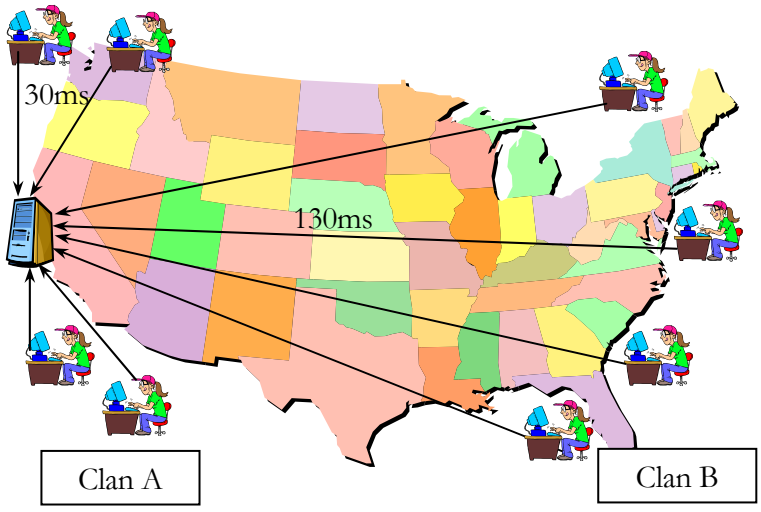

Figure 8: Today's static server placement creates unfairness for First-Person-Shooter Clan matches.

The scenario illustrated in Figure 8 shows what happens in today's world of statically placed first-person-shooter game servers, where the unfairness in delay between the participants results in the game state updates reaching the high-delay players significantly after the low-delay players.

This occasion arises frequently when clans (teams) from different countries are to play each other; it can also happen within large countries such as the US, as illustrated. In this scenario, clan A will typically get low pings to the server, on the order of 30ms round-trip time, while clan B will get pings on the order of 130ms or more. This leads to a ping differential of 100ms, which allows clan A players to see the future of clan B players, and thus, they can start shooting at them before clan B players detect their presence.

In such situations, it is not uncommon for clan B to call off the match as most gaming organizing committees (e.g., Cyber Athlete League [31]) stipulate that ping differences of more than 90ms can lead to voiding the match. The clans struggle to find an equi-ping server, which is not always possible because they need to have administrative access to that server to set-up the proper match configuration.

It is important to note again that unfairness is not due to the large absolute value of the ping, but rather due to the large ping differential between the participants. This delay fairness problem can be solved if we could position the game server hosting the match at an equal latency point with respect to the participants. However, it is also possible to try to achieve fairness by artificially inflating the lag for all participants [91], but this will lead to excessive sluggishness (e.g., the player shoots their gun then some time later the bullet hole appears on the wall across).

The game servers are usually statically separated in this manner because it is very hard for one clan to own more than one server in different places spread across

the Internet. There are a number of new renting services that allow clans to rent a server for a few days, but these are still not very economical since they require at least one day of rent, as opposed to a few hours for a single match. In addition, these renting services do not currently reveal much about the server location and whether it will be fair for all the participants.

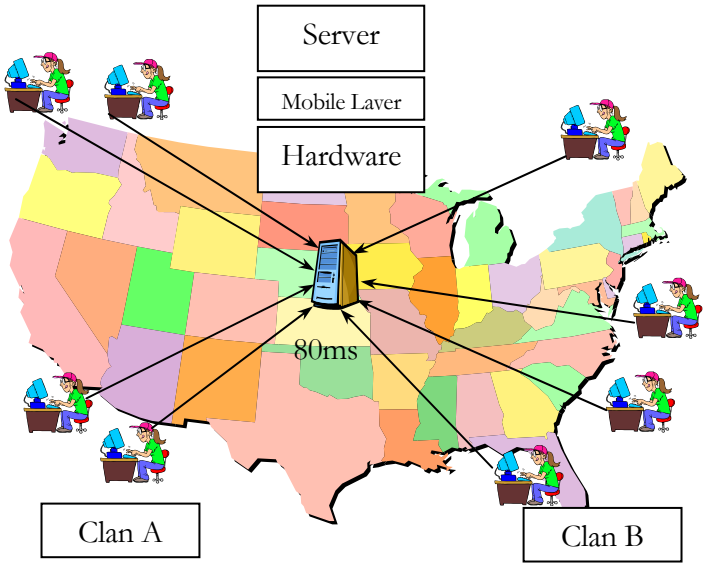

Figure 9: Dynamic game server placement allows placement of the server at an equi-ping node with respect to all participants.

If we can dynamically move the game server (carrying the OS and application code), then we can allocate an equi-ping hardware server in advance and move that game server to that location. In contrast to the same snapshot we presented in Figure 8, which led to a ping differential of 100ms, Figure 9 illustrates how we can have an equi-ping of 80ms for all participants, bringing the ping differential down to zero. In Chapter 6 we present how the vMatrix provides a solution for dynamically moving game servers to equi-ping locations.

2.1.5. Current Solutions for Application Mobility

There are a number of current solutions that attempt to address the Internet application challenges that we covered in sections 2.1.2. , 2.1.3. , and 2.1.4. . These solutions have a common drawback, which is a lack of backward-compatibility. In this section, we briefly cover these solutions, but we will discuss related work in more detail in the following chapters as it pertains to each of the three applications we developed on top of the vMatrix platform.

Application servers, such as ATG Dynamo [16], IBM WebSphere [51], BEA WebLogic [21], and JBoss [56] , provide a strict API for system services, and thus, it is feasible to move the application between different servers running the same application server. However, it is common that developers do not strictly adhere to these APIs (for performance and functional reasons), which prevents application mobility.

Java servlets [94] can be moved between servers that are running the Java virtual machine, but this approach suffers from performance degradation due to the real-time byte-code translation that Java requires. It also requires that existing applications be rewritten within the Java environment, which presents a high switching cost for already deployed applications.

Package manger tools, such as Debian's APT (Advanced Packaging Tool [35]) or Red Hat's RPM [85], can facilitate the installation/de-installation of Internet service code between servers; however, they do not provide any kind of isolation and library version collisions can happen between services installed on the same machine.

Disk imaging (aka ghosting) and diskless workstations booting from SANs have been used for years to quickly repurpose hardware to new services. However, that

approach suffers from the inability to concurrently run more than one VM per RM, which is necessary so that software developers can be presented with the same dedicated machine isolation model that they are familiar with.

OS virtualization, e.g., VServer [3][106], Ejasent [40], Ensim [41], and Denali [13], traps all OS calls, thus allowing applications to be moved across virtual operating systems. The downside of this solution is that it is OS-dependent, performance isolation is not guaranteed, and it imposes strict guidelines on what the applications can and can't do.

To our knowledge, no one has tackled the problem of optimal server placement for the purpose of reducing ping differential for first-person-shooter game servers. However, in section 6.4 we cover many previous research that addresses (1) other important aspects of having a distributed, large-scale, multi-player network of servers, (2) artificially inflating pings to achieve fairness, and (3) modeling game server traffic patterns so that ISPs can properly pre-provision network bandwidth for gaming services.

2.2 Virtual Machine Monitors

A fundamental component of the vMatrix architecture is Virtual Machine Monitors (VMMs). In this section, we give a brief overview of VMMs, focusing specifically on the functions relevant to our solution.

A VMM is a thin software indirection-layer that runs on top of a real machine and exports a partitioned abstracted view of that machine [82]. This abstraction is a virtualized (mimicked) view of all hardware in the machine (e.g., CPU, Memory, I/O), as shown in Figure 10.

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. 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.

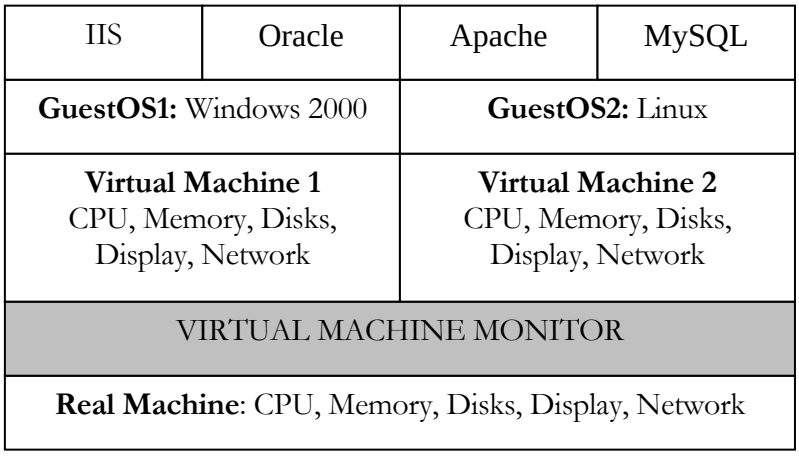

Figure 10: Virtual Machine Monitor.

Unlike a Java Virtual Machine [98], 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 thus, the performance of code running inside a VM is almost as fast as the code running directly in an RM.

Note that, in this work, we do not attempt to build a VMM, but rather we use existing software for the x86 architecture from VMware, Inc. [104] to build an overlay network of VMMs.

The core properties of VMMs that we rely on are:

(1) Backward Compatibility: Since the VMM runs on top of the hardware layer, old or new operating-systems/software-stacks are directly supported.

(2) Transferability: The encapsulated state of a VM can be moved from one real server running the VMM software to another.

(3) Isolation: Two VMs running on the same VMM are safely isolated from each other.

(4) Suspend/Resume: The VMM can suspend the hosted VM in a pre-booted state and resume it quickly when needed.

In the following sections, we cite relevant information from an overview article [68] in which Mendel Rosenblum and Tal Garfinkel elegantly described the technical implementation details of VMMs, specifically as they relate to the VMware VMM, which we use as the foundation building block of the vMatrix framework. Please refer to the full article [68] if you need more details regarding how VMware tackles virtualization limitations for current hardware and software systems.

2.2.1. Why the Revival of Virtual Machines?

Ironically, the capabilities of modern operating systems and the drop in hardware cost—the very combination that had obviated the use of VMMs during the 1980s—began to cause problems that researchers thought VMMs might solve. Less expensive hardware had led to a proliferation of machines, but these

machines were often underused and incurred significant space and management overhead.

To mitigate the effects of system crashes and security break-ins, system administrators again resorted to a computing model with one application running per OS/machine. This, in turn, increased hardware requirements, imposing significant cost and management overhead. Moving applications that once ran on many physical machines into virtual machines and consolidating those virtual machines onto just a few physical platforms increased use efficiency and reduced space and management costs. Thus, the VMM's ability to serve as a means of multiplexing hardware—this time in the name of server consolidation and utility computing—again brought it to prominence.

2.2.2. Decoupling Hardware from Software

As Figure 10 shows, the VMM decouples the software from the hardware by forming a level of indirection between the software running in the virtual machine (layer above the VMM) and the hardware. This level of indirection lets the VMM exert tremendous control over how guest operating systems (GuestOSs) use hardware resources.

A VMM provides a uniform view of underlying hardware, making machines from different vendors with different I/O subsystems look the same, which means that virtual machines can run on any available computer. Thus, rather than worrying about individual machines with tightly coupled hardware and software dependencies, administrators can view hardware simply as a pool of resources that can run arbitrary services on demand.

Because the VMM also offers complete encapsulation of a virtual machine's software state, the VMM layer can map and remap virtual machines to available

hardware resources at will and even migrate virtual machines across machines. Load balancing among a collection of machines thus becomes trivial, and there is a robust model for dealing with hardware failures or for scaling systems. When a computer fails and must go offline, or when a new machine comes online, the VMM layer can simply remap virtual machines accordingly. Virtual machines are also easy to replicate, which lets administrators bring new services online as needed.

The strong isolation model that VMMs provide is very valuable for reliability and security. Applications that previously ran together on one machine can now separate into different virtual machines. If one application crashes the operating system because of a bug, the other applications are isolated from this fault and can continue running undisturbed. Further, if attackers compromise a single application, the attack is confined to just the compromised virtual machine.

Thus, VMMs are a tool for restructuring systems to enhance robustness and security—without imposing the space or management overhead that would be required if applications were executed on separate physical machines.

2.2.3. VMM Hibernation and Remote Control

An important VMM feature that we use extensively in this work is the ability to suspend the guest VMs in a live state, such that all CPU registers, I/O buffers, and memory are dumped to disk, then the machine could be resumed later at the same point at which it was suspended (this is similar to hibernating a laptop). This feature has the added advantage that the suspended VMs can be resumed in a very short time, typically around 10 seconds, rather than booting up the machine from a cold state, which tends to take more time and consume more resources, and thus, improving recovery times significantly.

This encapsulation also provides for a very general mobility model, since we can now copy the suspended virtual machine files over a network (or a USB memory stick), which is one of the core VMM advantages that we leverage in the vMatrix architecture.

Most VMM software also provides remote control over the keyboard, monitor, mouse, and other I/O devices 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).

2.2.4. CPU Virtualization

The VMM must be able to export a hardware interface to the software in a virtual machine that is roughly equivalent to raw hardware and simultaneously maintain control of the machine and retain the ability to interpose on hardware access.

A CPU architecture is virtualizable if it supports the basic VMM technique of direct execution— executing the virtual machine on the real machine, while letting the VMM retain ultimate control of the CPU.

Implementing basic direct execution requires running the virtual machine's privileged (operating-system kernel) and unprivileged code in the CPU's unprivileged mode, while the VMM runs in privileged mode. Thus, when the virtual machine attempts to perform a privileged operation, the CPU traps into the VMM, which emulates the privileged operation on the virtual machine state that the VMM manages.

The VMM handling of an instruction that disables interrupts provides a good example. Letting a guest operating system disable interrupts would not be safe,

since the VMM could not regain control of the CPU. Instead, the VMM would trap the operation to disable interrupts and then record that interrupts were disabled for that virtual machine. The VMM would then postpone delivering subsequent interrupts to the virtual machine until it re-enables interrupts.

Consequently, the key to providing virtualizable architecture is to provide trap semantics that let a VMM safely, transparently, and directly use the CPU to execute the virtual machine. With these semantics, the VMM can use direct execution to create the illusion of a normal physical machine for the software running inside the virtual machine.

2.2.5. Memory virtualization

The traditional implementation technique for virtualizing memory is to have the VMM maintain a shadow of the virtual machine's memory-management data structure. This data structure, the shadow page table, lets the VMM precisely control which pages of the machine's memory are available to a virtual machine.

When the operating system running in a virtual machine establishes a mapping in its page table, the VMM detects the changes and establishes a mapping in the corresponding shadow page table entry that points to the actual page location in the hardware memory. When the virtual machine is executing, the hardware uses the shadow page table for memory translation so that the VMM can always control what memory each virtual machine is using.

Like a traditional operating system's virtual memory subsystems, the VMM can page the virtual machine to a disk so that the memory allocated to virtual machines can exceed the hardware's physical memory size. Because this effectively lets the VMM over-commit the machine memory, the virtual machine

workload requires less hardware. The VMM can dynamically control how much memory each virtual machine receives according to what it needs.

2.2.6. I/O virtualization

Thirty years ago, the I/O subsystems of IBM mainframes used a channel-based architecture, in which access to the I/O devices was through communication with a separate channel processor. By using a channel processor, the VMM could safely export I/O device access directly to the virtual machine. The result was a very low virtualization overhead for I/O. Rather than communicating with the device using traps into the VMM, the software in the virtual machine could directly read and write the device. This approach worked well for the I/O devices of that time, such as text terminals, disks, card readers, and card punchers.

However, current computing environments, with their richer and more diverse collection of I/O devices, make virtualizing I/O much more difficult. The x86-based computing environments support a huge collection of I/O devices from different vendors with different programming interfaces.

Luckily, industry trends in I/O subsystems point toward hardware support for high-performance I/O device virtualization. Discrete I/O devices, such as the standard x86 PC keyboard controller and IDE disk controllers that date back to the original IBM PC, are giving way to channel-like I/O devices, such as USB and SCSI. Like the IBM mainframe I/O channels, these I/O interfaces greatly ease implementation complexity and reduce virtualization overhead.

Chapter 3 THE VMATRIX FRAMEWORK

In this chapter, we provide an overview of the vMatrix architecture, and we will offer more implementation details in subsequent chapters as it relates to the different applications. The vMatrix is a network of real machines (RMs) running virtual machine monitor software (VMMs), such that virtual machine (VM) files 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).

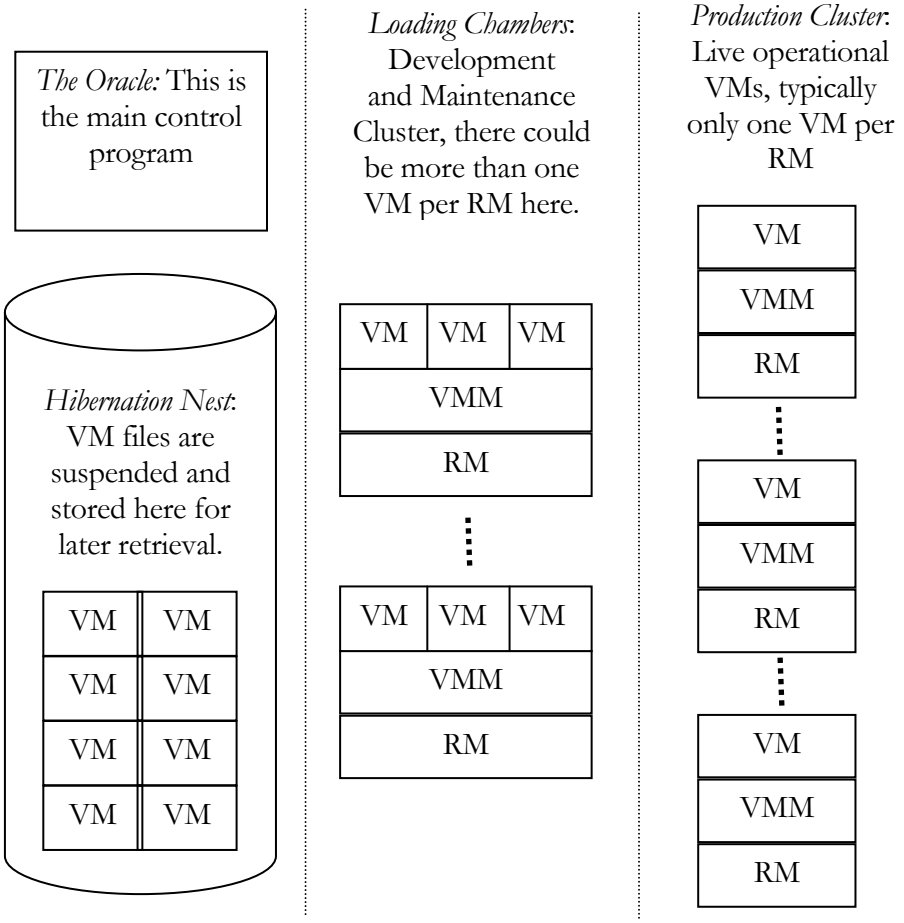

Figure 11: The vMatrix Framework.

3.1 Main Components

The basic framework for the vMatrix is illustrated in Figure 11. There are three 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 tier application servers, or the back-end databases. The important distinction is that, in this state, the VMs are exposed to operational load and there is only one VM per RM.
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, and upgrading libraries, etc. In this state, we can have more than one VM sharing the same RM, since the VMs are not exposed to live production load. The VMMs 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 is their own fully assigned, isolated, real machine. However, if these machines are exposed to heavy load, such as 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 quota scheduling system that prevents one VM from cannibalizing all of the RM resources (i.e., CPU, Memory, Disk space, I/O, Network, etc).
3. **The Hibernation Nest:** this is simply the back-end storage for keeping all the VM files in a dormant suspended state. The VMs are not accessible

in this mode, but can be quickly retrieved and instantiated onto either one of the other two clusters.

The Oracle is the control program responsible for maintaining the state of all VMs and RMs, and it supervises the vMatrix network. The main functions of the Oracle are:

(1) **Keep a registry of available RMs**: As new RMs are added to the network and loaded with the VMM software, they are subscribed with the Oracle. The Oracle continues to monitor and update the state of the RMs in the RM registry.

(2) **Keep a registry of VMs:** Whenever a new VM is created, it is registered with the Oracle, and the Oracle continues to maintain the state of that VM in the registry (i.e., is the VM currently active? Which RM is it assigned to?).

(3) **Match the VMs to the RMs:** The Oracle is responsible for the matching of VMs to RMs based on resource requirements and accrued load profiles. In section 5.2.2. , we provide more details on how the Oracle builds the VM to RM load profiles and performs the matching operation.

(4) **Transfer and Activation:** The Oracle is responsible for copying the VM files to that specific RM, then instructing the associated VMM to activate it.

(5) **Network Configuration:** The Oracle is responsible for configuring the network elements properly so that the VM can function properly at its

new location. In section 4.3 , we provide more details on how the Oracle configures the network environment around the VM.

In our 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 (using `scp` and a private key enabling access to the VMMs' host OS), 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 A).

3.2 VM Server Lifecycle

The simple state diagram shown in Figure 12 describes the life cycle of a VM Server:

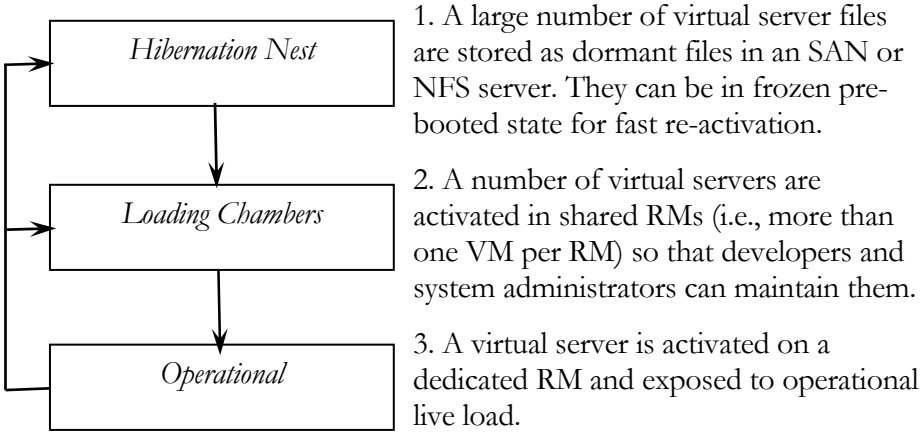

Figure 12: Life Cycle of a Virtual Server.

3.3 Basic Steps for Migrating a VM

These are the specific steps that our prototype Oracle took to move a VM into an RM. Note that these steps are dependent on the specific VMM that we used (i.e., VMware).

1) **Create the VM:** The first step is to create a new VM representing the service. The VM is created with the proper hardware configuration, then the new operating system is installed in it (as it would be installed inside an RM). The application code is then installed and the VM is registered with the Oracle.

2) **Suspend the VM**: Once the VM is fully booted and ready to accept operational load, the Oracle suspends it. This is achieved by sending the `suspend()` command to the hosting VMM, which freezes the VM and dumps the memory states into a `.vmss` file.

3) **Move the VM files:** The next step is to move the necessary VM files from the loading chambers' VMM (or the hibernation nest) to the operational VMM. The Oracle simply issues the `scp` command as follows:

3.1) **Copy the disk image file:**

```
scp oracle@rm1:/vmfx/vm1.dsk oracle@rm2:/vmfs/vm1
```

3.2) **Copy the suspended state file:**

```
scp oracle@rm1:/home/oracle/vmware/vm1/vm1.vmss
oracle@rm2:/home/oracle/vmware/vm1
```

3.3) **Copy the configuration file:**

```
scp oracle@rm1:/home/oracle/vmware/vm1/vm1.cfg
oracle@rm2:/home/oracle/vmware/vm1
```

4) **Register the VM**: Once the files are fully transferred, then Oracle registers this VM with the new hosting VMM; this is done by issuing the `register()` command while pointing to the `.cfg` file containing the configuration of this VM..

5) **Resume the VM**: The Oracle then resumes the VM by sending the `resume()` command to the hosting VMM.

6) **Update Load Balancer**: Finally, the Oracle adds the IP address of the new VM to the load balancer representing the VIP (Virtual IP address) for this service. This can be done using SNMP or similar means.

3.4 On-Demand VM Replication

On-demand VM replication is very useful for absorbing flash crowds and reducing failure recovery times.

On-demand replication can be achieved by one of two methods:

(1) **VM Cloning**: By this we mean that a copy of the VM file is made in real time and instantiated on a new RM. The disadvantage of cloning is that it is not always achievable without some changes to the service architecture or code (e.g., need to change the IP address for the clone, although NAT or DHCP can be used for that, as we will describe in section 4.3). For multi-tier architectures, as well, 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 make any code changes to the existing application code, we can pre-create all the server VMs needed for the worst-case scenario in the Loading Chambers, and then shut down and store all those VMs in the Hibernation Nest. When demand surges, the Oracle now has a large pool of VMs that it 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 (VM sizes range from a few megabytes to multi gigabytes, depending on the amount of software and data they contain), but this is not a big issue with the ever-decreasing storage costs. Furthermore, smart differential compression techniques (e.g., chain coding [30]) 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 of this solution is that the system administrators now have to manage all these VMs (e.g., if there is a new service patch, then it will have to be applied to all the VMs by activating them in the Loading Chambers or applying a differential patch at the VM level).

We chose the second approach because of 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 I/O buffers are dumped to disk. The machine then could be resumed later at the same checkpoint at which it was suspended (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, rather than booting up the machine from

an idle state, which requires a long time and consumes more resources. Thus, by pre-booting the service VMs, before suspending and storing them in the Hibernation Nest, when a flash crowd arrives, the Oracle 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 a 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.

Finally, the quick resumption of VM files from a suspended state offers improved availability, as a new VM can be instantiated fairly quickly to take over from a VM that failed due to a software crash, thus significantly reducing recovery time. In [15] , 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 A, we list the VMware Perl function to check for heartbeats from VMs to make sure they did not crash.

Chapter 4 DYNAMIC CONTENT DISTRIBUTION

4.1 Introduction

In this chapter, we propose and analyze the vMatrix as a practical solution for the distribution and replication of dynamic content. In a nutshell, dynamic content is the result of a program executed on the web server (e.g., a CGI script), as contrasted with static content, which is simply a file retrieved, as from the hard disk of the web server (e.g., an image).

Today, there are many solutions for distributing and replicating static content, ranging from classic web proxy caches [54][12][37][27] to the more recent content distribution networks (CDNs) [5][26][110]. The main benefits of distribution and replication are: improving end user response time; higher availability; absorbing flash crowds; and network bandwidth savings.

Recent Internet measurements indicate that non-cached dynamic content constitutes about 40% of web requests [102][8]. Dynamic content is indeed becoming more prevalent on the web as more applications are being ported from the desktop to become web services, which allows quicker bug fixing, upgrading, and collaboration, among other benefits.

However, the problem of distribution and replication of dynamic content continues to elude us due to the many dependencies that such services have on custom libraries, third party modules, operating systems, and server hardware. Simply copying the code of the dynamic service is not possible since the target

machines must have exactly the same environment for the code to run unchanged, which is not very practical. Previous solutions in the area of dynamic content distribution are not backward-compatible and require changes to the existing applications.

The main challenge that we focus on in this chapter is how to mimic the environment surrounding the virtual machine such that it can be moved transparently between real machines without making significant architecture or software changes. We demonstrate that, through the use of off-the-shelf technologies, it is indeed possible to mimic the surrounding environment. Through network address translation (NAT), it is possible to provide transparent mobility of VMs between RMs, such that the transported machine can resume normal operation once it is re-instantiated at a new location without the necessity to reconfigure its software (e.g., networking stack). For multi-tier architectures, virtual private networks (VPNs) allow us to provide secure connectivity between the transported front-end VMs and any other back-end machines they depend on.

We claim that the distinguishing advantages of this approach are: the very low switching cost of converting an existing web service to run within such a framework; the ability to move front-end servers to the edge; and the on-demand replication of servers to absorb sudden surges of requests. The main disadvantage is that the mobile VM files can be quite large, on the order of gigabytes; however, it is similar to large multimedia video files for which many solutions exist today for efficient delivery on the Internet [61].

In section 4.2 , we enumerate the main functional and performance challenges. In section 4.3 , we focus on how to mimic the environment surrounding a VM to allow transparent mobility and two-tier secure connectivity. In section 4.4 , we

analyze the performance implications of porting two-tier architectures into our proposed framework. Finally in sections 4.5 and 4.6 , we cover future work and related work and then offer conclusions.

4.2 Proposed Solution

Recall that the vMatrix is a backward-compatible solution that builds on top of the classic operating system concept of a Virtual Machine Monitor [82]. The observation we make is that a VMM virtualizes the real machine (RM) at the hardware layer (CPU, Memory, I/O), and exports a virtual machine (VM), which mimics exactly 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 encapsulation solves the dependency problems since the whole machine is transferred with the code, modules, libraries, and operating system that the dynamic service depends upon. Hence, the problem of server mobility is reduced to delivering large VM files within a network of RMs running the VMM. If there is a sudden surge in demand at any place in the world, then the web server hosting the service, or even the entire cluster, can be replicated and transported in real time (on the order of minutes) to be closer to the demand.

In contrast to previous work (which we cover in section 4.5), we claim that the vMatrix presents the smallest switching cost for porting an existing web service into such a dynamic content distribution network. In fact, single-tier architectures could be ported to this network with no code changes; the developer would simply need to install the service inside a VM, the same way he/she would install it inside an RM today. Once that is accomplished, the VM is ready to be instantiated anywhere in the world. Similarly, two-tier architectures could be

ported with no code changes, but might require some code changes for performance reasons, as discussed in section 4.4 .

In this thesis, we do not tackle the problem of where to send dynamic content, but rather, we tackle the problem of how to distribute and replicate dynamic content with minimal code changes. The problem of where to send dynamic content is very similar to the caching and distribution of static content, and the basic premise is to detect Internet demand hot spots, and then, cache content in close proximity to these hot spots. There are a number of solutions and algorithms for determining the best places to replicate content and mirror servers [5][93][92][67][71].

Achieving backward compatibility and fluid distribution does not come without challenges. The main challenges are twofold:

4.2.1. Functional Challenges

The main functional challenge is to mimic everything around the operating system and web service application such that it runs just as it would run in its native environment. There are two main components.

4.2.1.1. Machine Mimicking

As discussed in section 2.2 , the VMware VMM is able to entirely fool the guest code into believing its mimicked environment such that it cannot detect whether it is running inside a virtual machine or a real machine.

4.2.1.2. Network Environment Mimicking

Once the machine is mimicked, we then need to mimic its interactions with the outside world. We can always view any web server as a black box with the main

connectivity to the outside world achieved through its networking card. The trick then is to maintain the outside view from the machine regardless of where it is located. In section 4.3.1. , we cover, in detail, our approach to network environment mimicking, which is the dynamic reconfiguration of the networking devices around the VM so that it continues to function transparently in its new location. Our approach for network environment mimicking builds on off-the-shelf technologies, such as network address translation (NAT) and configurable virtual private networks (VPNs).

4.2.2. Performance Challenges

One of our goals is to allow services to be ported into this framework with minimal application changes. However, certain changes might be required to improve performance of the transported service. For example, if there is a large amount of communication between the front-end mobile VM and a back-end small database, then it might be best to move the back-end database with the front-end server, which could lead to significant code changes. Performance challenges are discussed in detail in section 4.4 .

4.3 Global-Local Architecture for Server Distribution

Recall from Chapter 3 that the vMatrix architecture is an interconnected mesh of RMs ready to host VMs. The production cluster RMs exist within entities that we refer to as hosting networks (HNs). HNs are Internet service providers that operate the RMs and make them available for hosting VMs.

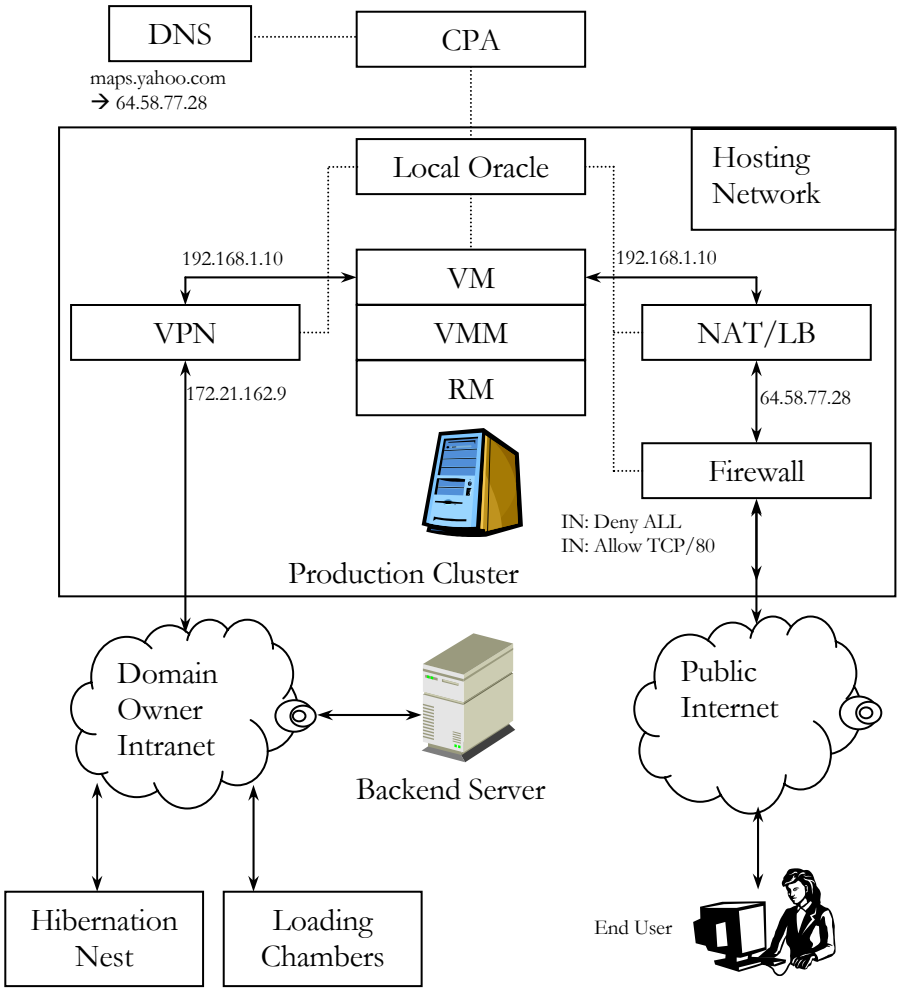

Figure 13: The Content Placement Agent (CPA) instructs the Local Oracle (LO) to instantiate a VM on a RM. The LO then configures the necessary networking components (NAT/Load Balancer, DNS, Firewall, VPN).

For the purpose of Dynamic Content Distribution on a global scale, we expanded the vMatrix into a two-level global-local architecture. Global assignment decides

to which HN a given VM should be sent, and then local assignment is responsible for choosing a suitable RM within that HN. Figure 13 illustrates the main components, which are:

1. **Content Placement Agent (CPA):** The CPA decides to which HN the content VM server should be sent to be in close proximity to the end users consuming content from that server. In its most basic form, its functionality is very similar to that of the distribution and replication of static content. Note that the CPA can also be a human operator observing a world map were he/she can see all locations with available real servers and then adjust a knob to increase the number of instances of any Internet service at any location with a demand hot spot.
2. **Local Oracle (LO):** The LO is similar to the Oracle we described in Chapter 3 , except that it runs locally in the HN and is only responsible for the matching of VMs to RMs within that HN. The LO is also responsible for configuring all the networking components (DNS, VPN, NAT/Load Balancer, Firewall) to maintain the mimicked network environment around the VM. Below we describe the functions of the LO in more detail.

At a high level, once the CPA decides that a new server VM needs to be instantiated within a given HN, it contacts the LO within that network. The LO then replies back whether there are available RMs to host the new VM. If so, then the CPA sends the VM file to the LO, and consequently, the LO transfers the VM file to a suitable RM, configures all the network components, and starts the VM server.

4.3.1. Network Mimicking and Transparent Global Mobility

By transparent global mobility, we mean the ability to move the VM server globally and have it resume normal operation at a new location without the need to reconfigure its software (e.g., networking stack or application-specific code). Our solution is based on network address translation techniques (NAT)[63]. The IP address for the mobile VM is pre-picked from a pool of private intranet IP addresses [112], e.g., the 192.168.x.x range. Machines belonging to the same domain (e.g., yahoo.com) should not use the same IP address more than once, to avoid conflicts between VMs instantiated in the same network. However, VMs belonging to different domains (e.g., yahoo.com and msn.com) can share the same private IP address since they will be mapped to different VLANs, as described in the next section.

Referring to Figure 13, the IP address chosen for the VM is 192.168.1.10. Once the LO instantiates the VM on the RM, it configures the NAT to map the internal private address, 192.168.1.10, to a public IP address of 64.58.77.28. The LO then contacts the CPA and instructs it to configure the primary DNS (Domain Name Server) to include the new IP address of 65.58.77.28 as a possible target for maps.yahoo.com (which we used as an example of a dynamic content mapping service in section 2.1.2.).

When a given user browser queries the network for maps.yahoo.com, the request is relayed to the primary DNS server for this domain, which uses the source IP address of the user's machine to translate maps.yahoo.com into the server IP that is closest to the user. Such a DNS response usually has a very short time-to-live attached to it, so that intermediate DNS servers relay the requests back to the primary DNS server. One known shortcoming of this approach is that, sometimes, intermediate DNS servers are configured to enforce a minimum time-

to-live, overriding the time-to-live specified by the primary DNS server. Similarly, some web browsers on the user side might cache the DNS response rather than resubmit the DNS query.

The main benefit of this approach for transparent mobility is that the VMs can be moved closer to the end users without the need to reconfigure them; instead, the LO configures the networking elements surrounding them, and the LO need not know any of the details of what is running inside the VM. This approach also allows cloning and replicating VM servers in different networks, since local private IP addresses are re-usable in distinct networks. Thus, if there is a full cluster in the East coast, and a spike of demand is detected in the Midwest, then the whole cluster could be cloned (essentially copies of the VM files) and replicated as is in the Midwest.

Note that the NAT box might also be a load balancer (LB). In this case, multiple VMs, with private IPs sitting behind the LB, can map to the same public IP address (VIP). The LO simply needs to add the private internal IP address for the new VM server to the rotation of private IPs serving the public IP address of 64.58.77.28. The load balancer then spreads incoming requests across all private IP addresses currently in rotation, including the new VM server that was just instantiated.

Another value added feature is absorbing flash-crowds. In that case, a variable number of suspended VMs could be resumed on demand, to absorb this short demand period. Once the flood is over, then the VMs can be suspended back to a dormant state, and the host RMs can be used for some other service. If a VM file is already present in the network with the surge of demand, then starting a new VM from a suspended state takes on the order of 10 seconds, compared to booting a server from scratch, which can take a few minutes.

Another issue that should be considered is moving or suspending servers with active web connections. The graceful solution is to take such servers out of load-balancer rotation, or remove their DNS and NAT mappings. Then, after determining that all existing connections are closed, the server can be safely suspended, moved to a new RM, and then resumed.

Although DHCP [81] is a possible alternative to NAT, it is not transparent to the software inside the VM and requires that the IP lease be renewed once the machine is moved to a new network. This dictates that the LO must be aware of which operating system is installed inside the VM and issue the right command sequence to invoke a renewal of the IP lease, versus the NAT approach, which is completely external to the VM. Another complication is that some server applications (e.g., ftp) within the VM server might cache the IP address, and changing the IP address using DHCP will require restarting all such applications to get the new IP address.

One possible work-around for renewing the DHCP IP lease without communicating directly with the hosted operating system is to instruct the VMM to disconnect the virtual NIC (network interface card) for the hosted VM. This would typically force the OS to re-issue a DHCP request and renew the IP lease.

Finally, for security reasons, the LO also configures a firewall to secure the public address. Thus, as shown in Figure 13, only port 80 will be allowed incoming access to the 64.58.77.28 public IP address.

4.3.2. Secure Connectivity

Two-tier architectures require that the mobile front-end VMs maintain connectivity to the back-end servers that they rely on. We propose a solution based on configurable virtual private networks (VPNs) [19]. In detail, once the

LO instantiates the VM on the RM, a VPN is configured, providing a secure IP tunnel (IPSec) [76] from the front-end VMs in the HN to the back-end servers in the intranet of the owner domain. Note that the VPN set-up is done transparently, below the VM; thus, it does not require any software configuration inside the VM. This is illustrated in Figure 13, where the 192.168.1.10 private address on one end of the secure tunnel is mapped to the 172.21.162.9 private intranet address (in the intranet of the domain owner) on the other end of the secure tunnel. The meta information attached with each VM file is the server local IP address and an authentication certificate for establishing the VPN tunnel.

Another security issue that should be considered is VM servers that belong to different domains but are connected to the same network switch (e.g., yahoo.com and msn.com). In this case, it is possible for the VMs to snoop on each other's packets or even attack each other. This problem could be solved by allowing the LO to configure VLANs (virtual LANs) on the network switch that the VMs are connected to, such that different domains belong to different VLANs. This provides perfect network isolation between domains, while also providing secure connectivity between VMs that belong to the same domain. Even in the case where two different domain VMs end up in the same RM, the VMM can provide separate VLANs for the hosted VMs that would prevent them from snooping or attacking each other.

4.4 Two-Tier Architectures

As discussed earlier in section 2.1.2. , single-tier architectures fit very nicely within the vMatrix; however, two-tier (and consequently n-tier) architectures have the added complexity of requiring communication from the front-end mobile VM servers to back-end servers (e.g., an email service requires connectivity to backend storage to fetch the user's email messages).

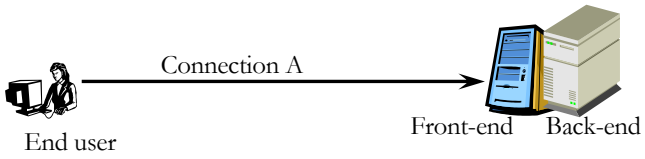

Figure 14: Non-Distributed Case: Two-tier architecture with front-end and back-end in same local area network.

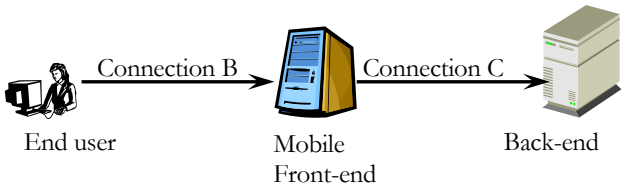

Figure 15: Distributed Case: Two-tier architecture with front-end servers distributed to be closer to end users.

Figure 14 illustrates a standard non-distributed two-tier architecture with both the front-end and back-end servers located in the same network, while Figure 15 illustrates a two-tier architecture with distributed mobile front-ends. The request from the end user is sent over connection B (which can be a number of Internet hops) to the mobile front-end machine, which then queries some information from the back-end database over connection C (again possibly a number of hops away), and then uses the returned results to generate a web page for the end user.

Referring to section 2.1.2. , the primary goals of content distribution are: faster response time, improved availability, on-demand replication, and saving bandwidth. The first three goals are clearly achieved in the single-tier case, and the fourth goal is achieved if the total number of bytes served by the VM server is larger than its size. However, as we will discuss in the following sections, for the two-tier case, the benefits may or may not be achieved, depending on the traffic patterns from the end user to the front-ends and from the front-ends to the back-ends.

4.4.1. Response Time

Here we present a back-of-the-envelope analysis to demonstrate that the response time benefits of front-end server distribution can still be achieved for the two-tier case under certain practical conditions.

In Figure 14, the response time perceived by the end user is primarily dominated by connection A. The communication overhead between the front-end and the back-end is relatively negligible, since it usually occurs over a well-provisioned 100Mbps switch or even a 1Gbps switch. However, in Figure 15, the response time is a function of both connection B and connection C. Let's assume that TCP is used for communication over connections A, B, and C. According to [70], a simplified formula to derive the response time (R) to serve a request is:

$$R \geq k \cdot N \cdot RTT \cdot \sqrt{p}$$

where k is a constant, N is the number of packets (derived as the request size divided by the maximum segment size allowed over the connection), RTT is the round trip time, and p is the probability of packet loss.

For front-end server distribution to be worthwhile, we require that the total response time in the distributed case (Figure 15) be less than the response time in the non-distributed case (Figure 14); thus,

$$N_B \cdot RTT_B \cdot \sqrt{p_B} + N_C \cdot RTT_C \cdot \sqrt{p_C} < N_A \cdot RTT_A \cdot \sqrt{p_A}$$

but N_B is equal to N_A, so the generalized requirement is:

$$N_C / N_B < (RTT_A \cdot \sqrt{p_A} - RTT_B \cdot \sqrt{p_B}) / (RTT_C \cdot \sqrt{p_C})$$

To further simplify this formula, assume that the front-end is moved as close to the end user as possible, such that (RTT_B, $\sqrt{p_B}$) will be relatively negligible, and (RTT_C, $\sqrt{p_C}$) will be almost equal to (RTT_A, $\sqrt{p_A}$); hence, the requirement reduces to:

$$N_C < N_B$$

Although this analysis is based on a TCP connectivity assumption, the final result is rather intuitive. For a benefit to be derived from moving the front-ends as close as possible to the end users, the number of packets exchanged between the front-ends and the back-ends should be smaller than the number of packets exchanged between the end user and the front-ends. However, some proprietary back-end protocols might make the assumption of communication over a highly responsive low-loss LAN, and thus, might be very sensitive to loss. When such protocols are used over a WAN, they might stall in the event of repetitive packet loss or long round trip times. Such protocols are not compatible with our proposed framework and will need to be rewritten under the assumption of WAN communication, i.e., they need to be resilient in case of frequent packet loss and large delays. It should be noted, though, that proprietary protocols are used in a very small number of websites, compared with TCP.

We also note that even in the case where $N_C > N_B$, , the end user might perceive better response time by recoding the front-ends such that the web page is partially rendered and a progress meter is displayed while the request between the front-end and back-end is fulfilled. The user perceives such an experience as more responsive than a very long delay before any parts of the web page are rendered due to a distant front-end.

Another solution to address cases where $N_C > N_B$ is to move the back-end database with the front-end machines. For example, if the back-end database is a relatively small read-only static geographical maps database, then we can package it in a VM and ship it in conjunction with the front-ends. This dictates the expanded notion of moving a cluster of VMs together, rather than individually. However, this solution is not practical for huge back-end databases that are hosted in a RAID system (on the order of 100GB), and it is also not practical when the end user is submitting information that needs to be written to the back-end servers, e.g., an auctions listing.

4.4.2. Availability

For the single-tier case, availability is certainly improved since the servers are geographically dispersed in more than one location. Hence, if connectivity is lost to one of the servers, or if that specific server is down, then it is always possible to find an alternative server. Simulation results by [18], confirm that distributing code inside the network results in an order of magnitude improvement of availability.

However, in the two-tier case, while availability between the end users and the front-ends is improved, availability between the front-ends and the back-ends is compromised. In Figure 14, the front-ends and back-ends are collocated on the

same network switch, which is usually a reliable system consisting of a primary switch with a hot standby to take over in case of failure. However, in Figure 15, the front-ends are connected to the back-ends over an unreliable WAN network and connectivity could be lost at any time. Most web services will display an error message to the end user when connectivity is lost to the back-end. Thus, at least in the distributed case, the end user will get a message informing them of the service interruption, versus when connectivity is lost to the front-ends, in which case, the user gets no indication of status, which is quite a poor user experience.

To further improve service availability in the distributed case, the front-ends can be re-designed to cache recent requests to the back-ends and use such cached results when connectivity is lost. Indeed, most web services are designed with this in mind since it provides a graceful degradation path. This solution only works for services that are read-only, or write-once read-many. For example, when the front-ends simply need to connect to a back-end ad server, then, by caching these ads on the front-ends, the ad can be re-used and the user will not feel the disconnection that took place. Similarly, user profiles for personalized web pages are typically write-once read-many, and hence, if the front-ends cache the user profiles then loss of connectivity will not affect the experience of the end user. The only downside of that is that the user might get a slightly stale profile, versus not getting any service if connectivity is lost to the front-ends.

As discussed above, in some scenarios, it might be reasonable to package the back-end database in a VM and ship it in conjunction with the front-ends, thus eliminating the availability problem between the front-ends and back-ends.

4.4.3. On-Demand Replication

For the single-tier case, multiple front-ends can be cloned (i.e., same VM file copied more than once) and replicated on-demand in different networks to absorb flash crowd requests. For the two-tier case, if the back-end databases allow any front-end to connect to them through some known port (e.g., Oracle's TNS listener), then we can certainly instantiate more front-ends as demand dictates. However, if the connectivity protocol between the front-ends and back-ends is a proprietary protocol that makes assumptions about the IP addresses or some logical name of the front-ends, then architecture changes will be required.

4.4.4. Bandwidth Savings

It is worth noting that the size of VM files will typically be on the order of gigabytes, even in compressed form. A 4GB file takes around 15 minutes to be transmitted over a typical T3 link, or 1 minute over an OC12 link, which is reasonable to achieve real-time distribution of dynamic content. This problem is very similar to the problem of transferring large multimedia video files for which many solutions exist today that provide efficient and reliable delivery [61].

However, we always need to weigh the 4GB that will be consumed while moving the server, against the bandwidth savings that will be achieved by moving the server closer to end users. If the total number of request bytes served by the web server, during its lifetime at the new location, is larger than the size of the VM file, then obviously bandwidth savings are achieved. It is also possible to pre-push the VM file of the server to all major networks smoothly overnight, when bandwidth is available, rather than abruptly during daytime hours. In addition, if the server is being replicated in a number of networks, then proper reliable

multicast techniques [61][33] can be used to further reduce the consumed bandwidth.

4.5 Related Work

The problem of static content distribution and replication is an inherently easy one, since the fundamental object being cached is a nicely encapsulated file that does not have many dependencies. Static content changes, at most, once every couple of days (e.g., images, fixed HTML pages, download files, video files). The common distribution and replication solutions are standard proxy cache hierarchies with expiration times associated with the cached objects [54][12][37][27], or CDNs that push such objects to the edge of the network [5][26][110].

It is important to distinguish the difference between frequently changing content as opposed to dynamic content. Frequently-changing content gets stale very quickly as time passes and must be delivered to all edge nodes in a real-time fashion (e.g., news, stock quotes, sports scores). It is sometimes also referred to as dynamic content. The common solutions for frequently-changing content distribution are partial/delta web page composition at edge nodes by pulling of content pieces from back-end servers [7][44][43], origin servers that track content dependencies and proactively push content to edge nodes [58][59][17][97][105][33][48], or a combination push-pull approach [78].

The main focus of this chapter is the distribution and replication of dynamic content. We define dynamic content as the result of a program executing on the server at the time of the request. Unlike dynamic content caching research that focuses primarily on saving server CPU cycles [17][23][102], our focus is to achieve the benefits covered in section 2.1.2.

Previous work in the area of dynamic content caching suffers from a common disadvantage, which is requiring the web 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.

The Internet Content Adaptation Protocol (ICAP) [52] is a light weight protocol, which allows remote procedure calls through HTTP so that servers can instruct proxy caches to do adaptation processing on the content (e.g., shrink an image). Active Cache [79] is a solution that allows servers to supply cache applets with the documents such that the proxy caches must run these applets on the documents before serving them (e.g., add a banner ad). The main complexity is how to write the code such that it is executable under different proxies from different vendors without compromising performance or security. Active cache uses a sand-boxed Java environment. Both Xenoservers [36] and Denali [13] require developers to write their code under a specialized operating system that is optimized for encapsulation and migration.

Application Servers [16][51][21] provide a strict API for system services, and thus, 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. Java servlets [94] can be moved between servers running the Java virtual machine, but this approach suffers from performance degradation, which might not be justifiable for a web service. It also requires that existing applications be rewritten within the Java environment, which presents a high switching cost. The Portable Channel Representation is an

XML/RDF data model that encapsulates operating system and library dependencies to facilitate the copying of a service [22] across different systems. Again, it requires the programmers to learn a new framework and port their existing work in to it, but it also does not provide isolation between servers that belong to different domains.

Operating system virtualization, e.g., VServer [3][106], Ejasent [40], and Ensim [41], traps all operating system calls, thus allowing applications to be moved across virtual operating systems. The downside of this solution is that it is operating system-dependent, performance isolation is not guaranteed, and it imposes strict guidelines on what the applications can and cannot do.

Finally, recent advances in XHTML, XML, CSS, DOM, and Javascript technologies led to the AJAX [57] programming model, which allows for a majority of the application logic to be downloaded in real time to the client's side. In this model, the data still resides on the back-end servers, but most of the application logic is downloaded to the client upon the first connection, and then, the data is fetched in pieces as needed from the server, and the browser web page is updated dynamically. This places the application as close as possible to the user, and leads to extremely interactive applications that almost match resident desktop applications. Some examples of AJAX applications are Internet services, such as Google Maps and the new Yahoo Mail. Another widely used technology that allows for real-time downloadable applications is Adobe's Flash platform [2], and the new Yahoo Maps is an example of that.

4.6 Conclusion

In this chapter, we described how the vMatrix framework provides a backward-compatible solution for the distribution and replication of dynamic content. We illustrated that, using off the shelf technologies, we could mimic the network environment surrounding the virtual machine such that it can be transparently moved without the need to reconfigure its software. We also discussed the performance challenges to fitting existing two-tier services within such a framework.

Chapter 5 SERVER SWITCHING

5.1 Introduction

In this chapter, we analyze the vMatrix practical solution for sharing a pool of servers between a number of Internet services with varying load profiles. The idea is analogous to the well-known computer networking trade-off between circuit-switching and packet-switching. Today, most Internet services are provisioned in a circuit-switching-like fashion, i.e., 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 potentially the waste of much capacity if the service does not use 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, but rather, falls within some probabilistic bound, since if all of the services peak at the same time, and there was no adequate buffer of idle servers, then there might not be enough servers to accommodate all the services, causing congestion, and, eventually, server-loss (i.e., demand for another server, but there are no severs available).

Currently, server switching is primarily possible within a few standardized application server frameworks, such as ATG Dynamo, IBM WebSphere, and BEA WebLogic [16][51][21], and even within those frameworks, library versions and operating system release mismatches can lead to inter-operability problems. In addition, 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 in 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 often than not, Internet service infrastructures grow in an evolutionary fashion rather than a revolutionary one.

The main reason for the difficulty in 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 must 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 run on the same server.

The vMatrix solves the software dependencies problem because the whole service is transferred with the OS, libraries, code, modules, and code that the service depends upon. 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), thus mimicking the same hardware environment for that service regardless of the real hardware of the hosting real machine (though there might be

performance degradation). But, most importantly, vMatrix solves the people dependency problem by presenting the developers and system administrators with the same isolation model with which they are familiar, with statically allocated servers.

In this chapter, we present ways in which the vMatrix can be used to load-balance the virtual machine services across real machines to maximize utilization efficiency (in terms of machines and people costs) such that the 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, which allows 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 they are familiar with;

(3) Backward compatibility, which results in very low costs for 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 [107].

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

In section 5.2 , we discuss the vMatrix implementation details as they relate to server switching. In section 5.3 , we discuss our experiences with two Internet services migrated to the vMatrix platform. Finally, in sections 5.4 and 5.5 , we cover related work and offer conclusions.

5.2 Implementation Details

In contrast to previous work (which we cover in section 5.4), we claim that the vMatrix 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, maintaining most of the key advantages of static pre-allocation described in section 2.1.3. (mainly isolating software, people, and hardware dependencies). In section 5.3 , we illustrate this ease of conversion through a few real-life examples.

5.2.1. 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 a VM, the same way they install it inside an RM today. Once that is accomplished, the VM is ready to be instantiated on any RM running the VMM software.

Referring to Figure 11 in Chapter 3 , the VM preparation and software installation is performed in the Loading Chambers, where the RM's 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.

5.2.2. 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 achieved by building a time-based profile for how many 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 I/O and network utilization). Appendix A illustrates the VMware Perl function to return the resources consumed by a given VM averaged over the last five minutes.

In this thesis, we make a simplifying assumption of one-to-one matching for operational VMs, i.e., only one VM per RM. This is a logical assumption since the operational VMs are typically under heavy load and need a dedicated RM (the Loading Chambers can continue to have many VMs per RM since they are non-operational). This simplification reduces the problem to a simple bottleneck detection and greedy matching algorithm.

In a nutshell, the Oracle loops over all operational VMs for a given service and detects those with a persistent bottleneck (e.g., 90% CPU utilization over the last 10 minutes). The Oracle then fetches another VM for that service from the Loading Chambers and activates it on an idle RM. Conversely, if there are no bottlenecks detected for any of the operational VMs for a given service, then the Oracle should move back one of the VMs for that service 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, in which case, we would still need the switching algorithm to detect and respond fairly quickly.

Finally, the server-switching algorithm takes into account the VM sizes to minimize VM switching so that the data center LAN is not overloaded with VM transfers, although this is now less of an issue with the ever-increasing LAN network speeds or if the VM images are operated directly from a SAN.

An issue that should be considered is de-activating VMs with active connections. The graceful solution is taking such servers out of load-balancer rotation, and, after detecting that all existing connections are closed, safely suspending and removing these servers from the RM (this is actually very similar to how hardware servers are added and removed in a static solution, but this method is much faster and does not require humans to touch the physical machines).

It should be noted that VMware now offers VMotion [103] technology that can move live 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 will not work). However, the advantage of VMotion is that it can migrate live servers in less than two seconds by performing clever memory delta paging using bitmaps.

5.3 Experiences

It is the goal of this work to show that it is possible to encapsulate legacy Internet services via VMMs, and 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 a vMatrix prototype, and porting into it a number of existing Internet services, ranging from open-source services (e.g., PHPnuke [77] and osCommerce [113]) 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, such as troubleshooting, rebooting, monitoring, and code updates, etc.

5.3.1. The Experimental Set-up

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%.

5.3.2. A Web Portal: PHP-Nuke and osCommerce

PHP-Nuke is one of the most popular web content publishing open-source platforms that are written using the popular PHP web scripting language. It provides many functionalities for a full-fledged portal, such as news, polls, and message boards, etc. osCommerce is another popular PHP application that provides an e-commerce 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 16. 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.

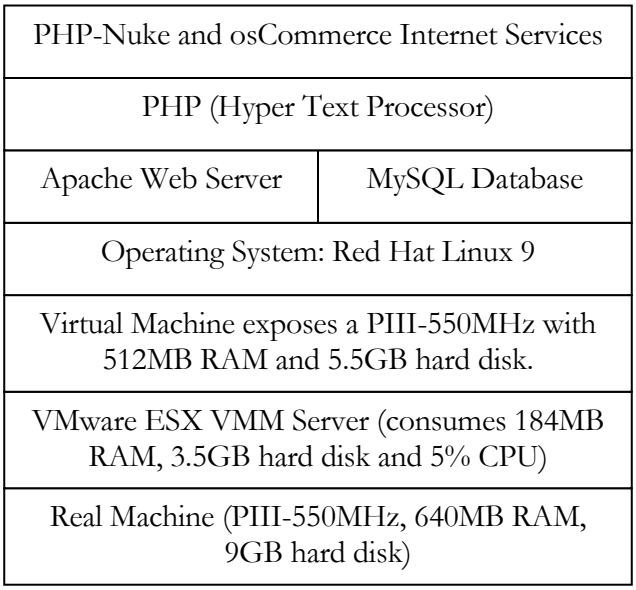

Figure 16: VM for PHP-Nuke and osCommerce.

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 and resumed. Note that when a VM is suspended in a pre-booted state, only three files need to be copied. The first is the configuration file for the VM, describing its memory size, and 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 three 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 whether they have crashed; however, this is simply a redundant check since most websites already have more sophisticated pings in place using monitoring tools, such as Nagios [42].

An expansion of this service that we could not accomplish 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 back-end 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.

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 17: VM for Yahoo! Autos Search.

5.3.3. Yahoo! Autos Search

Yahoo! Autos allows users to search for cars being sold from a number of sources. In this part of the experiment, we took the Yahoo! Autos Search functionality and installed it within the vMatrix framework, as illustrated in Figure 17. This is a typical Yahoo! Autos Search back-end 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.). The server then 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 consists of custom Yahoo C/C++ libraries. Most of the Yahoo servers use FreeBSD (rather than 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 the previous section, the Yahoo! Autos Search service was installed within the vMatrix framework in about a few hours, and no coding changes whatsoever 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 used in this set-up was to lie to the underlying VM as to how much physical memory was really present (so that we could match the memory requirements). Although 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. However, if the active working set of the VM truly needs 1024MB of physical memory then it will run slower due to this virtualization, thus this is 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 is an illustration of the ease of converting an existing service into this framework without the necessity for any coding or architectural changes. However, we attempted to give brief estimates in Figure 16 and Figure 17, which illustrate that the CPU performance overhead is usually about 5%, and the memory overhead is about 184MB. In addition, 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. Thus, 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.

5.4 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, such as ATG Dynamo [16], IBM WebSphere [51], BEA WebLogic [21], and JBoss [56], provide a strict API for system services, and thus, 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. Application Servers also fail to provide the strict isolation model that developers expect from a dedicated machine. Java servlets [94] 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. This model also requires that existing applications be rewritten within the Java environment, which again presents a high switching cost. Similarly, Denali [13] requires developers to write their code under a specialized OS optimized for encapsulation and migration.

The Portable Channel Representation [22] is an XML/RDF data model that encapsulates OS and library dependencies to facilitate the copying of a service across different systems. Again, this 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 manager tools, such as Debian's APT (Advanced Packaging Tool [35]) or Red Hat's RPM [85], 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 the same machine. Computing on the Edge [72] also fails in this category and suffers from the same disadvantages.

Disk imaging (aka ghosting) and diskless workstations booting from SANs have been used for years to quickly repurpose hardware to new services. However, that approach suffers from the inability to concurrently run more than one VM per RM, which is required 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 familiar with.

OS virtualization, e.g., VServer [3][106], Ejasent [40], and Ensim [41], traps all OS calls, thus allowing applications to be moved across virtual operating systems. The downside of this solution is that it is OS-dependent, performance isolation is not guaranteed, and it imposes strict guidelines on what the applications can and cannot do. Zap [90] sits somewhere between OS virtualization and application virtualization, but does share the same downside of being tied to the OS.

Finally, it should 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 [49], SUN provides the N1 system [95], and HP has the Utility Data Center [47].

5.5 Conclusion

In this chapter we presented the ways in which the vMatrix framework enables server switching. 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.

Chapter 6 EQUI-PING SERVER PLACEMENT FOR ONLINE FPS GAMES

6.1 Introduction

In this chapter, we describe how the vMatrix can enable fair, online, first-person shooter (FPS) gaming. One of the most important aspects for participants of online multi-player games is to have equal round-trip time to the server hosting the match (also known as ping time). As Figure 18 shows, different classes of online games have different tolerance to lag. Since lag is dependent on how fast the action in the game progresses, FPS games are the most sensitive.

Turn Based Strategy (Civilization 4, Risk, Chess, Pool)	**Not sensitive to lag** (lag difference on the order of seconds)
MMORPG and RTS (World of Warcraft, Everquest, Starcraft)	**Moderate sensitivity to lag** (lag difference on the order of 500ms)
First Person Shooters (FPS) (Halo, Battlefield, Doom, CounterStrike)	**Extremely Lag sensitive** (lag difference on the order of 50ms)

Figure 18: Lag sensitivity of different multi-player game classes.

Lag is very crucial since it dictates the frequency of state updates from the host to the clients; thus, the players with lower pings can indeed see the future of lagging players and shoot them before they know it Not only that, the lagging players have incorrect positions for the other players as their clients try to extrapolate the

current state based on the last (now stale) state update they got from the server (a common technique known as Dead Reckoning). Professional players are very aware of this limitation, thus explaining the reason why they take zigzag paths while running around to fool the extrapolation performed by the gaming clients. This forces the other players to shoot at false projected positions rather than their actual positions, which leads the game server to register a miss.

Previous studies in this area demonstrated that players prefer absolute pings to be less than 180ms, and some players further emphasized that relative delays between participants is more important than the absolute value of latency to the server [46][45][99][80][101].

Currently, most game servers are pre-located at static nodes around the Internet, which is adequate for the casual game where players join and leave all the time (they just pick the server closest to them). However, serious players usually form what is called a clan, and they purchase or rent a hosting server so that they can host the clan matches on it. Typically, each clan gets a server local to their country, or their area of the country (in case of large countries like the US). Thus, for example, in the US, an East coast-based clan will tend to get a server in the East coast, while a West cost clan will get a server in the West coast. This leads to the unfairness issue when the clans have matches against each other. The East coast clan will argue that the match should happen on their East coast server, and the West coast clan will argue for the reverse. The reason arguments arise is that typical round-trip-times rise from less than 30ms when playing on a server in the same coast as the clan players, to more than 130ms when crossing over to the other coast. This difference of 100ms gives a large advantage to the local clan; they literarily see the positions of other players and shoot at them before they get there. These differences can be much worse than 100ms for cross-country matches.

It is important to re-iterate that it is not the large round-trip-time (also known as ping) that irks players the most, rather, it is the difference in ping that leads to the unfairness. Currently deployed solutions to this problem are unsatisfactory. For example, one solution is alternating matches between the home and away team, but this still does not change the fact that, during each match, the low-ping team will have a significant advantage. Another solution is searching for another clan with a server central to both of the participating clans; however, this is very hit-or-miss, and most of the time, such a server cannot be found since one of the participating clans still has to have the password for the game server to set up the proper match configuration.

The solution is accessibility to game servers that are equi-ping to all participants. However, its very cost-prohibitive to install copies of each game on servers all over the Internet; rather, we need an economical solution that allows us to share the hardware easily between different games (and even other vMatrix applications that are not gaming-related).

The main reason game servers are difficult to move in and out of hardware servers is the dependencies that the server code has on operating systems, libraries, third party modules, server hardware, and even people (the game administrators). Simply copying the code of the game server is not possible since the target machines must have exactly the same environment for the code to run unchanged, which is not practical. The library versions and patches that work with one game server might cause another server to fail.

The vMatrix allows us to encapsulate the state of the entire game server in a VM file, which could then be instantiated on any RM running the VMM software, thus addressing the dependency problems as follows:

(1) It solves the software dependencies problem since the game server is transferred with the OS, libraries, code, modules, and code that it depends on.

(2) It solves the hardware dependencies problem since the VMM can lie to the overlying game server about the hardware resources available to it (e.g., memory size), thus mimicking the same hardware environment for that service regardless of the real hardware of the hosting real machine (though there might be performance degradation).

(3) It solves the people dependencies problem since the VM capsule can include the passwords and access rights that the game administrators need to manage the server and set up the proper matches.

Hence, the problem is reduced to equi-ping placement of large VM files within the vMatrix.

In this chapter, we address how to place the virtual machine game servers services across the real machines to minimize ping difference between the participating players without degrading the server performance (in terms of latency, throughput, and availability). Another challenge that we touch upon is how to avoid making any significant architecture or software changes to existing game servers so that this solution is backward-compatible.

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

1. Equi-ping placement of game servers to minimize round-trip time difference between participants;
2. Backward compatibility, which leads to zero cost for converting existing game servers to run within our framework;
3. Economies of scale by leveraging the fact that this network can be shared among many different types of games, rather than being fully dedicated to one game.

In section 6.2 , we discuss the vMatrix implementation details as they relate to equi-ping game server placement. In section 6.3 , we discuss our experience with migrating the Halo PC game server to the vMatrix platform. Finally, in sections 6.4 and 6.5 we cover related work and then offer conclusions.

6.2 Implementation Details

In contrast to previous work (which we cover in section 6.4), we claim that this solution presents the smallest switching cost for porting an existing game server into such a dynamic allocation network (i.e., backward compatibility) and, at the same time, achieving the advantages of equi-ping allocation and economies of scale. In section 6.3 , we illustrate this ease of conversion through a real-life example.

6.2.1. Equi-Ping Game Server Placement

The game server VMs must be instantiated on an RM that is equi-ping to all participants; however, participants are usually not known until match time. The

straightforward approach to obtain latency information is to build ping profiles from each participant to all available RM servers just before the match begins. This solution does not scale very well if we have a large pool of servers to choose from. A ping topography map for the Internet is required; such a database should be able to return the ping difference, given two IP addresses. Akamai's EdgeScape IP database [6] can return the connection speed for a given IP, but will not provide the latency between two given IPs. A more practical solution is Meridian [24] from Cornell University; this database can efficiently return the latency between a set of player nodes and a set of server nodes by using routing measurements and gossip protocols. King [66] is another tool that can be used to efficiently estimate the latency between two nodes using their immediate downstream DNS servers as proxies.

Thus, the first step in the placement algorithm is to build a cross matrix with the pings between each of the player IPs and the available RM IPs. The algorithm then proceeds as follows:

Let S be the set of RM servers available to host the match, and let m be the number of available servers ($m = |S|$).

Let P be the set of players participating in this match, and let n be the number of players ($n = |P|$).

Let $RTT_{s,p}$ be the round-trip-time (ping) from server s to player p, where s belongs to S, and p belongs to P.

Next, we eliminate all servers s that have $RTT_{s,p}$ larger than 180ms to any player p, which is the maximum ping that FPS players can reasonably tolerate. This

elimination leads to a smaller set, S', that only contains servers that satisfy the 180ms constraint.

Now, for each server s in S', we compute a closeness factor, C_s which represents the average differential ping between the players if server s is chosen:

$$C_s = \frac{\sum_{p \in P} \sum_{\substack{i \in P \\ i > p}} \left| RTT_{s,p} - RTT_{s,i} \right|}{n(n-1)/2}$$

(Note that the ping difference when i and p are the same player is zero, so we exclude that, also the difference between players i and p is the same as difference between players p and i, so we only need to count that once).

Last, we search for the smallest C_s among all S' servers and that is our target server to host the match.

The order of complexity of this algorithm is $O(n^2m)$, which is n^2 to compute each closeness factor for n players to a given server, then m to search for the smallest across all servers. Notice that this is not as expensive as it seems, since n is usually small (on the order of 16 to 64 players maximum), so this is really a linear algorithm of number of servers.

The placement decision can be cached in case the same set of players play again. The Oracle then proceeds to copy the VM image for the needed game server to that location. Once the VM is transferred, the Oracle instructs the VMM software to activate that image, thereby making the server available to the players.

Note that most VMM software allows for the suspension of VMs in a live state, such that all CPU registers, memory, and I/O buffers are dumped to disk; then

the machine could be resumed later at the same checkpoint at which it was suspended (this is similar to suspending/hibernating a laptop). Hence, by pre-booting the game server VMs, before suspending and storing them in the Hibernation Nest, when a match is about to start, we only need to copy the suspended VM to the RM and then resume it fairly quickly, with no need to wait for a full boot to occur. Once the match is over, then the VMs can be suspended back to a dormant state, and moved to the Loading Chambers (for maintenance) or Hibernation Nest (for storage), and the RM is now freed for a different game server.

VMware's VMotion [103] technology would enable us to further optimize the location of the game server while the match is progressing. However, the two-second lag introduced by VMotion will certainly be noticed by the players and might have adverse effects if it happens during a critical moment in the match. Recently published work also shows that hot migration is possible, based on the Xen VMM [29]. In that paper, the authors illustrated mobility for an active, first-person-shooter game server (Quake III), which manifested itself as a 60ms temporary jitter for the participating players. However, this solution also requires speedy access to the VM files via an iSCSI gigabit interface. Thus, both VMotion and Xen mobility are not suitable between data centers, but they can certainly be used to improve availability within a data center.

6.2.2. Backward Compatibility

Most game servers could be ported to this framework with minimal to no code or infrastructure changes; the game administrators would simply need to install the OS and game server software inside a VM, the same way they install it inside an RM today. Once that is accomplished, 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 RM's main purpose is to host many idle VMs so that administrators can prepare them for operational deployment. The VMs are not exposed to any operational load while waiting in the Loading Chambers, other than allowing the administrators to test their configurations.

Note that the 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 administrators are concerned, when they connect remotely to a given VM in the Loading Chambers, they truly believe it is their own fully-assigned, isolated real machine. However, if these machines are exposed to a heavy load, such as decompressing a large tar-ball, then neighboring administrators might sense a sudden slow-down, and can start to realize that they are sharing the machine with somebody else. It must be noted though that server-class VMM software alleviates this issue by providing a resources quota system that prevents VMs from cannibalizing all of the RM resources (i.e., CPU, memory, disk space, I/O, network, etc).

6.3 Experiences

It is the goal of this work to show that it is possible to encapsulate legacy game servers via VMMs, and to achieve a standardized solution for equi-ping placement without requiring cost-prohibitive changes to existing system architectures. We illustrate that this is a practical solution by building a vMatrix prototype, and porting into it Microsoft's popular Halo PC game server, which is currently a widely deployed game server.

Our experience confirms that the migration cost is negligible, i.e., no code changes, quick migration, and a short learning curve.

6.3.1. The Experimental Set-up

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%.

Halo PC Game Server (278MB)
Operating System: Windows XP (1.8GB)
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 19: VM for Microsoft's Halo PC Game Server.

Microsoft's Halo PC is one of the most popular first-person-shooter games. 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 19. The time it took us to achieve this was not significantly more than it would have taken to install on a normal real machine. We did not change a single line of source code for the game server (in fact we did not even have access to the source code, just the executables), and it became fully supported within the VMatrix framework as is.

Once we configured the VM for Halo PC in the Loading Chambers, we next instructed the Oracle to activate the VM, which caused the VM to be stopped, copied over to the operational cluster, and then restarted. When a VM is stopped, only two files need to be copied. The first is the configuration file for the VM that describes its memory size and Ethernet address, etc., and the second file represents the hard disk of the VM. Once the two files are copied over to a dedicated RM in the operational cluster, it is restarted and becomes ready to accept a live match. Note that the restart operation, although a bit time-consuming (as opposed to a suspend/resume operation), has the side benefit of forcing the game server to reregister itself with the gaming directory service (typically, GameSpy Arcade Host Directory [53]). This allows the game server name to be mapped correctly to the IP address at the new location, thus providing transparent mobility.

The resulting VM file size was about 2GB (1.8GB for Windows XP, 10MB for Halo PC Server executables, and 268MB for the environment maps). This translates to 1GB gzip compressed, which takes less than two minutes to transfer over 100Mbps Ethernet, or about four minutes over a T3 line. The transfer time can be further reduced by the use of smart differential compression techniques (e.g., chain coding [30]), although this might add some decompression overhead in pulling the VM files back from storage. Thus, the activation time to add servers can be very reasonable and on the order of a few minutes.

The experiment we have described here is limited, in that we performed it in a single lab at Stanford, and therefore, with mostly low LAN pings. We think it is relatively straightforward to generalize the experiment to perform cross-Atlantic matches, but we did not have access to an Internet-wide VMM cluster.

Note that performance analysis of VMware virtualization overheads is not the goal of these experiments (in fact VMware has strict guidelines against publishing explicit benchmarking metrics); rather the goal is to illustrate the ease of converting an existing game server into this framework without the need for any coding or architectural changes. However, we attempted to give rough estimates in Figure 19, which illustrates that the VMM CPU performance overhead is usually less than 5%, the memory overhead is about 184MB, and the hard disk overhead is about 3.5GB.

6.4 Related Work

To our knowledge, no investigators have tackled the problem of optimal server placement to reduce the ping differential for first-person-shooter game servers. However, there has been much previous research that has addressed: (1) other important aspects of having a distributed, large-scale, multi-player network of servers; (2) artificially inflating pings to achieve fairness; and (3) modeling game server traffic patterns so that ISPs can properly pre-provision network bandwidth for gaming services. In this section, we offer a quick overview of these solutions.

Reference [88] is the most related work among the papers we surveyed, as it tackles the problem of multi-player server selection (rather than placement). It presents an architecture by which to choose a game server that minimizes the lag differential between a group of players. It assumes a dense pre-deployed population of game servers, such that an optimal server can be selected; however, this is only true for the most popular FPS games.

Reference [91] proposes changing the game server to artificially inflate the lag of all players to make them equal. These authors present a very sound analysis of how the game server can track the error perceived by each player due to a stale

state, and then, how to compute the proper delay to hold back the state updates such that all players observe the same error. Although this technique certainly improves fairness, it penalizes the players with good connections (they do propose a budget scheme to try to mitigate this effect).

Reference [34], from the IBM TJ Watson Research Center, proposes an on-demand service platform for online games, which builds on top of standard grid technologies and protocols. The main issues they tried to address were: reducing latency; improving scalability; and achieving economies of scale by sharing the platform between multiple game servers. Though they also tackled the problem of first-person shooters, they did not attempt to directly address the issue of minimizing the ping differential between players to achieve fairness. Their solution is not fully backward-compatible; it requires game developer awareness of the service platform (i.e., requires code changes), although they tried to minimize that as much as possible.

Another paper from a separate group at IBM TJ Watson and Columbia University [64] proposes a zoom-in-zoom-out algorithm for adaptive server selection for large-scale interactive games. Their focus is to minimize resource utilization while still providing small latency to participants. Their algorithm is primarily targeted for MMORPGs (Massively Multi-player Online Role-Playing Games), which typically have relaxed delay differential requirements compared to FPS games (can tolerate up to seconds of delay differential versus a maximum of 180ms for first-person shooters). While MMORPGs do not exhibit significant cross-coast delay differential unfairness, they can still exhibit cross-continent unfairness, which is why game companies typically localize MMORPGs on a per-continent basis.

There are many solutions [32][4][87][20][25] similar to those reported in the two papers listed above, that either focus on the MMORPG problem and/or propose a new platform that requires significant code rewriting for the game servers.

Another branch of interesting research in this area focuses on modeling the traffic generated by game servers [109][100][60] so that ISPs and game server providers can properly provision their networks. Although this type of research helps reduce overall latency, it does not address the unfairness problem due to ping differentials.

It is worth noting that Halo 2 on the Xbox Live [75] platform uses a very interesting method to choose the host for the matches. Rather than having dedicated hosts dispersed on the Internet, the Halo 2 matching servers choose one of the participants to be the hosting server (in addition to being a client). The participant is chosen based on historical information that they collect about the throughput and availability of that player when chosen as host previously. The obvious downside of this solution is that the hosting player is always going to have the smallest ping, which gives them a significant advantage. This approach also has many security issues, as the players attempt to hack their local host code to cheat.

Finally, we refer the reader to the related-work in section 4.5 , which contrasts the VMM approach with other solutions, such as Application Servers, Java servlets, Packaging Tools, OS Virtualization, and Disk Imaging (also known as ghosting).

6.5 Conclusion

In this chapter, we presented ways in which the vMatrix can be used for equi-ping, first-person shooter game server placement, which reduces the delay differential between all participating players to achieve a fair match. We described our approach in detail and provided a real-life example based on Microsoft's popular Halo PC game. The main advantages of our approach are backward compatibility and the economies of scale that such a virtual machine network provides (since it can work with any game server without the need for code rewriting).

Chapter 7 CONCLUSIONS

In this thesis, we presented the vMatrix framework, which is an overlay network of virtual machine monitors. Virtual Machines decouple the service from the server, and then the vMatrix allows for the mobility of server VMs between servers in a global, shared, overlay network.

We demonstrated that it is possible to encapsulate legacy Internet services via VMMs to achieve a standardized solution for improving the scalability, interactivity, availability, and delay-fairness of web services and FPS multi-player games without requiring cost-prohibitive changes to existing system architectures. The main applications we demonstrated were:

Dynamic Content Distribution: Moving services closer to the Internet edge, thus reducing latency and rendering such services more interactive and available for end users;

Server Switching: Sharing a pool of servers between more than one service, thus leveraging the benefits of statistical multiplexing to reduce overall system cost;

Equi-ping Game Server Placement: Placing game servers at optimized locations to improve the delay differential fairness of multi-player, first-person shooter games.

These services represent a wide spectrum of practical Internet application architectures (e.g., single-tier, two-tier, write-once/read-many, write-many/read-many, single-user, one-user to many-users, and real-time many-to-many users). We illustrated that the migration cost is minimal for both the developers of the service and the system administrators (i.e., quick migration and short learning

curve), while providing continuity for traditional system administration tasks, such as troubleshooting, rebooting, monitoring, code updates, and log collection.

The distinguishing advantages of the vMatrix framework are backward compatibility and presenting system developers with the same development model that they are currently accustomed to. The potential disadvantage of the vMatrix framework is the virtualization overhead, specifically large VM files, but we presented possible solutions to mitigate these issues.

7.1 Future Work

There are a number of ideas that we did not have time to explore in this thesis but which could serve as possible explorations for other researchers in the same field, and these are:

Numerous VMs Per RMs Scheduling: In this thesis we made the assumption of one-to-one mapping between VMs and RMs in the operational environment; however, due to the distribution of the VM front-end servers closer to the users, and over a potentially large number of geographically dispersed RMs, it is very possible that the number of user requests to any given VM is at such a low rate that only a fraction of the resources of the hosting RM are used (CPU, memory, hard-disk, network bandwidth). In this case, it is desirable to instantiate more than one VM per RM to maximize the utilization of the RM resources. The key to achieving this is to balance the maximization of the utilization of the host RM, without compromising the performance of the guest VMs. Most VMM providers are already providing resource scheduling among multiple VMs, thus allowing for such an optimization.

Global Optimization: In section 4.3 , we assumed the presence of a content placement agent (CPA) that decides where dynamic content should be delivered. There are a number of solutions and algorithms for determining the best places to replicate content and mirror servers. However, due to the fluidity and small time-scale that our solution introduces, it is possible to move servers around much more quickly, and thus, static mirror placement algorithms might not provide the optimal placement [93][92][67]. Some previous solutions attempted to address the problem of dynamic server mobility [71]. We think it would be interesting to investigate a grander optimization problem that takes into account all of the following criteria:

1.Location of RMs where VMs can be instantiated;

2.Location of users sending requests to the servers;

3.Location of back-end servers relative to RMs;

4.Size of VM files relative to size of server requests;

5.Cost of bandwidth during different times of day.

Then, the output of such an algorithm would be the optimal time and location to instantiate a new server. Opus [84] presented one way to address this optimization problem.

Compute Utility: A special case of vMatrix is to provide a compute utility service, where the location of the VM servers is not important, but rather, the VMs simply need to consume compute cycles any place in the world to perform simulations or other computational tasks.

Peer-to-Peer vMatrix: In this thesis, we assumed that the vMatrix is centrally controlled by a single entity that maintains a registry of the RM server locations and their hardware configurations, etc. It would be interesting to investigate a peer-to-peer version of the vMatrix, where there was no

central entity, but rather, new users could join the collective network with their own RM server—in other words, a Bit-Torrent network for copying and hosting VMs.

APPENDIX A

This appendix 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");
```

BIBLIOGRAPHY

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

[2] "Adobe Flash Player", Adobe Inc. http://www.adobe.com/products/flash/about/

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, M. Wawrzoniak, "Operating System Support for Planetary-Scale Network Services", First Symposium on Networked Systems Design and Implementation (NSDI), California, USA, March 2004.

[4] A. R. Bharambe, S. Rao, and S. Seshan. "Mercury: A scalable publish-subscribe system for Internet games". Workshop on Network and System Support for Games (NETGAMES), Germany, April 2002.

[5] Akamai, Inc. "Akamai Content Distribution Network". http://www.akamai.com/html/en/sv/content_delivery.html

[6] Akamai, Inc. "Akamai EdgeScape: Internet IP knowledge base". http://www.akamai.com/en/html/services/edge_how_it_works.html

[7] Akamai, Inc. and Oracle Corporation. "Edge Side Includes". http://www.edge-delivery.org/

[8] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna R. Karlin and Henry M. Levy. "Organization-Based Analysis of Web-Object Sharing and Caching". USENIX Symposium on Internet Technologies and Systems, 1999.

[9] 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.

[10] Amr Awadallah and Mendel Rosenblum. "The vMatrix: Server Switching. IEEE 10th International Workshop on Future Trends in Distributed Computing Systems". Suzhou, China, May 2004.

[11] Amr Awadallah and Mendel Rosenblum. "The vMatrix: Equi-Ping Game Server Placement For Pre-Arranged First-Person-Shooter Multi-

player Matches". The 4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2006), Dubai/Sharjah, UAE, March 2006.

[12] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. "A hierarchical Internet object cache". Proceedings of the USENIX 1996.

[13] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. "Denali: A Scalable Isolation Kernel". Proceedings of the Tenth ACM SIGOPS European Workshop, France, Sept. 2002.

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

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

[16] Art Technology Group, Inc. "ATG Dynamo Application Server". http://www.atg.com/en/products/das.jhtml

[17] Arun Iyengar and Jim Challenger. "Improving Web Server Performance by Caching Dynamic Data". 1997 USENIX Symposium on Internet Technologies and Systems.

[18] B. Chandra, M. Dahlin, L. Gao and A. Nayate. "End-to-end WAN Service Availability, Third Usenix Symposium on Internet Technologies and Systems". March 2001.

[19] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. "RFC2764: A Framework For IP Based Virtual Private Networks". February 2000.

[20] B. Knutsson, H. Lu, W. Xu and B. Hopkins. "Peer-to-Peer Support for Massively Multi-player Games". INFOCOM, Mar. 2004.

[21] BEA Systems, Inc. "BEA Systems WebLogic Application Server". http://www.beasys.com/products/weblogic/server/index.shtml

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

[23] Ben Smith, Anurag Acharya, Tao Yang, and Huican Zhu. "Exploiting result equivalence in caching dynamic web content". Oct 1999, USENIX Symposium on Internet Technology and Systems.

[24] Bernard Wong, Aleksandrs Slivkins, and Emin G¨un Sirer. "Meridian: A Lightweight Network Location Service without Virtual Coordinates". ACM SIGCOMM 2005, Pennsylvania, USA, August 2005.

[25] Butterfly, Inc. "The Butterfly Grid". 2003. http://www.butterfly.net/platform

[26] Cable & Wireless. "Digital Island Content Distribution Network". http://www.digisle.net/services/cd/.

[27] Cache Flow, Inc. "Cache Flow Edge Accelerators". http://www.cacheflow.com/products/index.cfm/

[28] Carl Waldspurger, "Memory Resource Management in VMware ESX Server", Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI), December 2002.

[29] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live Migration of Virtual Machines". USENIX NSDI 2005.

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

[31] Cyber Athlete Professional League. "The Cyber Athlete Professional League". http://www.thecpl.com/league/

[32] D. Bauer, S. Rooney, and P. Scotton. "Network infrastructure for massively distributed games". Workshop on Network and System Support for Games (NETGAMES), Germany, April 2002.

[33] Dan Li and David R. Cheriton. "Scalable Web Caching of Frequently Updated Objects using Reliable Multicast". USENIX Symposium on Internet Technologies and Systems, 1999.

[34] Debanjan Saha, Sambit Sahu, and Anees Shaikh. "A Service Platform for Online Games". Proceedings of the 2nd workshop on Network and system support for games, May 2003.

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

[36] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. "Xenoservers; Accounted execution of untrusted code". IEEE Hot Topics in Operating Systems (HotOS) VII, March 1999.

[37] Duane Wessels et al. "Squid Internet Object Cache". http://squid.nlanr.net

[38] Duane Wessels, Traice Monk, k Claffy, and Hans-Werner Braun. "A distributed test bed for national information provisioning". http://ircache.nlanr.net/

[39] E. Bugnion, S. Devine, K. Govil, M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors", ACM Transactions on Computer Systems (TOCS), November 1997.

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

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

[42] Ethan Galstad. "Nagios: Open Source Host, Service and Network monitoring program". 1999. http://www.nagios.org

[43] Fred Douglis, Antonio Haro, and Michael Rabinovich. "HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching". USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA, December 1997.

[44] Gaurav Banga, Fred Douglis, and Michael Rabinovich. "Optimistic Deltas for WWW Latency Reduction". USENIX 1997.

[45] Grenville Armitage and Lawrence Stewart. "Limitations of using Real-World, Public Servers to Estimate Jitter Tolerance Of First Person Shooter Games", ACM SIGCHI ACE2004 conference, Singapore, June 2004

[46] Grenville Armitage. "An experimental estimation of latency sensitivity in multi-player Quake 3". 11th IEEE International Conference on Networks (ICON 2003), Australia, September 2003.

[47] Hewlett-Packard. "HP Utility Data Center". http://ww.hp.com/solutions1/infrastructure/solutions/utilitydata/

[48] Huican Zhu and Tao Yang. "Class-Based Cache Management for Dynamic Web Content". IEEE, Infocom 2001.

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

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

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

[52] ICAP. "Internet Content Adaptation Protocol". http://www.i-cap.org/icap/media/draft-elson-opes-icap-01.txt

[53] IGN Entertainment, Inc. "Game Spy Arcade Game Server Directory". http://www.gamespyarcade.com/

[54] Inktomi Corporation. "Inktomi Traffic Server". http://www.inktomi.com/products/cns/products/tseclass.html

[55] Inktomi Corporation. "Traffic Server Audited Performance Benchmark", 1998. http://www.inktomi.com/new/press/1998/inkbench98.htm.

[56] JBoss, Inc. "JBoss: Open Source Java Application Server (J2EE)". http://www.jboss.org

[57] Jesse James Garrett, "Ajax: A New Approach to Web Applications", Adaptive Path Publications, February 2005. http://www.adaptivepath.com/publications/essays/archives/000385.php

[58] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. "A Publishing System for Efficiently Creating Dynamic Web Content". INFOCOM 2000.

[59] Jim Challenger, Paul Dantzig, and Arun Iyengar. "A Scalable System for Consistently Caching Dynamic Web Data". 18th Annual Joint Conference of the IEEE Computer and communications Societies, 1999.

[60] Johanness Faerber. "Traffic Modeling for Fast Action Network Games". ACM Multimedia Tools and Applications, May 2004.

[61] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. "A Digital Fountain Approach to Reliable Distribution of Bulk Data". ACM SIGCOMM 1998.

[62] J. Sugerman, G. Venkitachalam, and B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", Proceedings USENIX Annual Technical Conference, June 2001.

[63] K. Egevang and P. Francis. "RFC1631: The IP Network Address Translator". May 1994.

[64] KangWon Lee, BongJun Ko, and Seraphin Calo. "Adaptive Server Selection for Large Scale Interactive Online Games". ACM NOSSDAV'04, Cork, Ireland June, 2004.

[65] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. "The Measured Performance of Content Distribution Networks". 5th International Web Caching and Content Delivery Workshop, May 2000.

[66] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. "King: Estimating Latency between Arbitrary Internet End Hosts". SIGCOMM Internet Measurement Workshop 2002, France, November 2002.

[67] L. Qiu, V. N. Padmanabhan, and G. Voelker. "On the Placement of Web Server Replicas". IEEE Infocom 2001.

[68] Mendel Rosenblum, and Tal Garfinkel. "Virtual Machine Monitors: Current Technology and Future Trends", IEEE Computer Magazine, May 2005.

[69] M. Kozuch and M. Satyanarayanan, "Internet Suspend/Resume", Proceedings IEEE Workshop Mobile Computing Systems and Applications, June 2002.

[70] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. "The macroscopic behavior of the TCP Congestion Avoidance algorithm". Computer Communications Review, July 1997.

[71] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. "A Dynamic Object Replication and Migration Protocol for Internet Hosting

Service". 19th IEEE International Conference on Distributed Computing Systems, Austin, Texas, June 1999.

[72] 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.

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

[74] Microsoft Corporation. "Microsoft Virtual Server". http://www.microsoft.com/windowsserversystem/virtualserver/default.mspx

[75] Microsoft Corporation. "Xbox Live". http://www.xbox.com/live

[76] P. Srisuresh. "RFC2709: Security Model with Tunnel-mode IPSec for NAT Domains". October 1999.

[77] PHPNuke.org. "PHP-Nuke: Open Source Advanced Content Management System". 2000. http://www.phpnuke.org

[78] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. "Adaptive Push-Pull: Disseminating Dynamic Web Data". Tenth International World Wide Web Conference, Hong Kong, May 2001.

[79] Pei Cao, Jin Zhang, and Kevin Beach. "Active Cache: Caching Dynamic Contents on the Web". Proceedings of Middleware 1998.

[80] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. "Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game". ACM SIGCOMM Workshop Network and System Support for Games (NETGAMES), 2004.

[81] R. Droms. "RFC2131: Dynamic Host Configuration Protocol". March 1997.

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

[83] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, Monica Lam, "The Collective: A Cache-Based System Management Architecture", Proceedings Symposium Network Systems Design and Implementation (NSDI), USENIX, May 2005.

[84] Rebecca Braynard, Dejan Kostic, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. "Opus: an Overlay Peer Utility Service". Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH), June 2002.

[85] Red Hat, Inc. "RPM: Red Hat's Package Manager". http://www.rpm.org

[86] R. Sites, A. Chernoff, M. Kirk, M. Marks, S. Robinson, "Binary Translation", Communications of the ACM, February 1993.

[87] S. Fiedler, M. Wallner, and M. Weber. "A communication architecture for massive multi-player games". Workshop on Network and System Support for Games (NETGAMES), Germany, April 2002.

[88] Steven Gargolinski, Christopher St. Pierre, and Mark Claypool. "Game Server Selection for Multiple Players". ACM NETGAMES 2005, New York, USA, October 2005.

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

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

[91] Sudhir Aggarwal, Hemant Banavar, Sarit Mukherjee, and Sampath Rangarajan. "Fairness in Dead Reckoning based Distributed Multi-player Games". ACM NETGAMES 2005, New York, USA, October 2005.

[92] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. "Constrained Mirror Placement on the Internet". IEEE Infocom 2001.

[93] Sugih Jamin, Cheng Jin, and Yixin Jin. "On the Placement of Internet Instrumentation". IEEE Infocom 2000,

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

[95] Sun Microsystems. "Sun N1". http://www.sun.com/n1

[96] Syam Gadde, Jeff Chase, and Michael Rabinovich. "Web Caching and Content Distribution: A View from the Interior". 5th International Web Caching and Content Delivery Workshop, May 2000.

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

[98] Tim Lindholm and Frank Yellin. "Java Virtual Machine Specification", The, Second Edition". 1999, Addison-Wesley. ISBN: 0201432943.

[99] Tom Beigbeder, Rory Couglan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. "The Effects of Loss and Latency on User Performance in Unreal Tournament 2003". ACM SIGCOMM Workshop Network and System Support for Games (NETGAMES), 2004.

[100] Tristan Henderson, and Saleem Bhatti. "Modeling user behavior in networked games". ACM Multimedia 2001, June 2001.

[101] Tristan Henderson. "Latency and User Behaviour on a Multi-player Game Server". Third International Workshop on Networked Group Communication (NGC2001), London, UK, November 2001.

[102] V. Holmedahl, B. Smith, and T. Yang. "Cooperative Caching of Dynamic Content on a Distributed Web Server". 7th IEEE International Symposium on High Performance Distributed Computing, 1998.

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

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

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

[106] VServer, "Linux-VServer: OS Virtualization for GNU/Linux Systems" http://linux-vserver.org/Welcome_to_Linux-VServer.org

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

[108] Wu-chang Feng and Wu-chi Feng. "On the geographic distribution of online game servers and players". Workshop on Network and System Support for Games (NETGAMES), 2003.

[109] Wu-chang Feng, Francis Chang, Wu-chi Feng, and Jonathan Walpole. "Provisioning Online Games: A Traffic Analysis of a Busy Counter-Strike Server," SIGCOMM Internet Measurement Workshop, November 2002.

[110] Xcelera, Inc. "Mirror Image InstaDelivery Internet services". http://www.mirror-image.com/

[111] XenSource, Inc. "Xen: An open source Virtual Machine Monitor for x86". http://www.xensource.com/index.html

[112] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. "RFC1918: Address Allocation for Private Intranets". February 1996.

[113] osCommerce. "osCommerce: Open Source eCommerce platform". http://www.oscommerce.com