Datalogisk Institut
KØBENHAVNS UNIVERSITET

DIKU

Virtual Machine Mobility With Self-Migration

Jacob Gorm Hansen

Technical Report no. 08-05
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

VIRTUAL MACHINE MOBILITY WITH SELF-MIGRATION

July 27, 2007
changeset: 147:e3c097ffbf3a

Ph.D. Thesis by
Jacob Gorm Hansen
Department of Computer Science,
University of Copenhagen

Preface

This work is a continuation of the NomadBIOS project, where Asger Jensen and I designed and built what would now be termed a para-virtualized hypervisor for hosting multiple virtual machines on top of the L4 microkernel, and—to our knowledge—were the first to show the feasibility of live migration of virtual machines across a local network. Focusing on grid computing, our idea was to package each grid job in a VM, and have these VMs roam the grid in search of idle resources. This work is described in our joint Master's thesis "Nomadic Operating Systems" [Hansen and Henriksen, 2002].

When starting my Ph.D.-studies about a year later, a group at the University of Cambridge had just released the first version of their own hypervisor, called Xen. Compared to the L4 microkernel on which our work was based, the Xen VMM had a number of advantages; it was simpler and faster, came with built-in drivers for up-to-date hardware, and because it was explicitly focused on hosting virtual machines, required less effort when porting or maintaining guest operating systems. Only a few weeks into my stipend, I travelled to Cambridge and started cooperating with the Xen team on bringing the live-migration features developed for NomadBIOS to Xen. While struggling to understand the design of their hypervisor, it occured to me that live VM migration would require a redesign of Xen to make it more like L4, compromising some of Xen's simplicity. Inspired by Steven Hand's "Self-paging in Nemesis" [Hand, 1999] paper, it occured to me that if a Xen guest OS could be self-paging, it also had to be possible for it to be *self-migrating*, able to migrate its state to another host, without outside help.

After a few months of experimentation, the first prototype implementation of a self-migrating operating system was functioning, and while my research has since strayed in other—still VM related—directions, self-migration and self-checkpointing have remained at the crux of my efforts. Live VM migration in the form first implemented in NomadBIOS, is now a popular technology, because it allows system administrators to reconfigure computing resources without inconveniencing users. In contrast, the interest in self-migration has been more academic, likely because its design goals—simplicity and security—provide only little benefit in the short-term, as large-scale VM deployments are still too rare to attract any significant hacking attacks. The topic of security has increasingly attracted my attention, so while my original research plan focused narrowly on creating a cluster management system built on the technologies developed for NomadBIOS, my recent work has attempted to address other problems, such as Desktop PC security, display multiplexing, and application persistence, as well.

Acknowledgements

The work described in this thesis would not have been possible without the support of several people and institutions. My Ph.D. stipend was funded by the University of Copenhagen Faculty of Sciences, and by the Danish Center for Grid Computing. My adviser, Professor Eric Jul, called on my cell phone while I was crossing into South Africa on a bus from Maputo, Mozambique, and arranged that I could fax in a handwritten application before the deadline. Eric has been a great guide to the possibilities and perils of the academic system, and has allowed me to tap into his worldwide network of top researchers in both academia and corporate research. The day after receiving the stipend, Eric booked me a ticket for SOSP in upstate New York that same weekend, and there he introduced me to Ian Pratt from the Cambridge University Computer Lab. Soon after, I was in Cambridge, working on VM migration in Xen. Keir Fraser helped me understand the innards of Xen, and Steven Hand proofread my first workshop paper. Paul Barham of Microsoft Research Cambridge hosted me as a research intern the following summer, and also put me in touch with Úlfar Erlingsson, from MSR Silicon Valley, whom I interned with during the summer of 2006. At MSRC I met and worked with a bunch of all-star graduate students, notably Andrew Phillips, Anish Mohamed, Suchi Saria, and Eno Thereska. Suchi invited me to a couple of Stanford lectures to get my head blown off, and the chance to hang out with Eno and his CMU crowd, including Niraj Tolia, is always sufficient motivation for going to systems conferences.

The people in the operating systems group at the Technical University of Dresden, led by Professor Dr. Hermann Härtig, were the first to notice our work on NomadBIOS and invite us for a visit, and the quality of the work done in Dresden was one of my main sources of motivation for pursuing the Ph.D. degree. I am also very grateful to Professor Ole Lehrmann Madsen and his OOSS group for hosting me at DAIMI in Aarhus.

Eric Jul is a graduate of the University of Washington, and I was fortunate to be able to spend six months there as a visitor, working with Eric's legendary adviser, Hank Levy, as well as Steven Gribble and Richard Cox in the fabulous environment at the UW. Ratul Mahajan introduced me to Seattle and graduate life there, and had all the qualities a good roommate should, including a motorbike and a mother who cooks Indian food.

In Copenhagen, Eske Christiansen has been a great help getting things running and general good company, and Jørgen Sværke Hansen has been a fun travel companion and discussion partner. Fritz Henglein indirectly suggested the name "Evil Man" for my project. The original work on VM migration was done with Asger Henriksen, who, in the process of changing his last name to Jensen, introduced me to Susanne, my wonderful wife. Susanne, thank you for always being patient with me, my travels, my working at strange hours, and my eternal indecisiveness, and thanks for being such a loving and caring person. I would also like to thank my parents and my brother Rasmus for letting me stay indoors with my computer all the time.

Other people I must thank, in completely random order, include Brian Vinter, Peter Andreasen, Annette and Dan, Benjamin Hindman, Andrew Warfield, Stephane Eisen, and everyone in the DIKU distributed systems group. Many other friends, relatives, and people in industry and academia, have inspired or influenced this work, and I am grateful to all of them.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

This thesis is about virtual machine (VM) mobility, and how it can be implemented without adding functionality to the virtual machine hosting environment. Our main contribution is the design and implementation of an algorithm that can live-migrate a running operating system between two physical hosts, with very little downtime, and without *any* support from the underlying virtual machine monitor or host environment. This is achieved by extending a commodity operating system with a *self-migration* mechanism, and through a number of supporting technologies, such as an unprivileged network boot-strapping mechanism for use in VMs.

Based on these technologies, we have designed and implemented a cluster-management system that supports job mobility, provides a high degree of flexibility, and is simple to manage. On top of this system we have been able to run real-life workloads, and have been experimenting with other interesting uses, such as rapid instantiation of jobs using hypercube network-forks. In parallel with this, we have also explored the use of VMs in desktop scenarios, and have successfully implemented live self-checkpointing of VMs running desktop workloads, including graphical applications with hardware-accelerated 3D content.

1.1 Motivation

The motivation for our work has been the emergence of grid- and cluster-computing, where hundreds or thousands of commodity computers work towards some common goal. This is a scenario that is not well handled by existing operating systems, that were originally designed for providing multiple trusted users with terminal access to a single physical machine, and not for untrusted users sharing access to large numbers of physical machines, connected by untrusted networks.

Grid computing [Foster et al., 2002] was born out of necessity; researchers in the natural sciences needed cheap computing power, which they obtained by combining large numbers of commodity PCs into *clusters*. Several clusters in turn were combined into *grids*, so that institutions could pool resources and increase utilization. The availability of these cheap computing resources motivated the development of new and improved scientific applications, resulting in increased demand.

Early grid software was developed *ad hoc* by physicists and chemists, by combining existing software packages such as batch scheduling systems, file transfer tools, distributed file and authentication systems, message passing and numerical libraries, into working setups. Whenever a new application depended on a new library or system-level service, it was simply added to the base install. This approach works well for small setups, but is hard to maintain across institutional boundaries, where coordinating and agreeing on a standard install may be difficult, or at least costly in terms of time and money spent on travel and meetings.

At the system level, the software running on grid nodes was often derived from UNIX [Ritchie and Thompson, 1974], and thus both lacking facilities that would be useful in a distributed environment, and burdened by features in conflict or redundant with those provided by grid middle-wares. For instance, to allow users to login and start jobs on a foreign cluster, a user identity would have to be created on each node in that cluster. Either this had to be done manually, or access control could be centralized using a service such as Kerberos [Miller et al., 1987]. Such a centralized service would require centralized control by one or more human operators, and become a bottleneck and single point of failure. When the user's job had run to completion, any files and processes created by the job would then have to be garbage-collected, to free up disk and memory space. Failing that, a human operator would have to occasionally log in and clean up the system. Existing grid systems are thus quite complex and error-prone.

We believe that by redesigning the system software running on grid nodes, we will be able to construct grid and on-demand systems that are simpler to develop, deploy, and manage. Ideally, there should be no need for each user to have an account on every grid node, and no need for a pre-installed set of applications libraries for the user's applications to depend upon. Ultimately, there should be no need for a human operator, and no need for a standards body to meet and discuss which libraries and services to supply with the base install.

What the user, or customer, really wants, is to run his job on a number of nodes, in a way that is as deterministic and dependable as possible. The input to each node should simply consist of the application and all of its software dependencies, as well as its input data. There is no need to authenticate the user at each node, only to authenticate the actual job itself.

Another problem with the use of existing operating systems as a foundation for grid computing, and the problem that we are primarily concerned with, is the lack of job mobility. Job submission and scheduling in existing grid systems is often centralized and static, and it is not uncommon for a user to have to wait several weeks before being able to submit even a short job, because already running jobs are monopolizing all nodes and cannot be preempted. Existing mobility systems fail to address this problem, by not being compatible with existing software, or because they suffer from residual dependency problems.

1.2 Problem we are Attacking

The traditional way of configuring a cluster is with a separate multi-user operating system, and a set of problem-specific application software packages, shared libraries, and service daemons, on each cluster node. If the cluster is supposed to run multiple tasks, for example, because it is shared by multiple users, different packages need to be installed. Herein lies the first problem as **name space collisions** are likely to arise, when different versions of the same library are required, if two service daemons need to bind to the same network port, or if different operating system kernel versions are needed. Other conflicts that are likely are that user names or numerical identifiers may collide, or simply that user A's application may consume too many resources, leading to **starvation** of user B's application. The final problem is the **lack of preemption and mobility** in cluster systems. After having run for a while, the cluster administrator (whether a real person or an automatic management daemon) may discover that the system load has become unbalanced, or that a long-running job is occupying a subset of the nodes and thus standing in the way of a short-running job that requires access to all nodes for a short time. This can be solved by completely purging the conflicting jobs, but at the cost of having to restart their execution from the beginning, losing any progress made. Just like in a traditional operating system, the ability to preempt and suspend a job momentarily, as well as the ability to migrate a job to a different CPU, is often useful because is adds flexibility and relaxes the need for perfect ahead-of-time resource scheduling decisions.

Preemption of jobs in a cluster has previously been attempted with the use of inter-node *process migration*. A process migration system captures the entire state of a running program, *i.e.*, all state in registers, main memory, and on disk, serializes it, copies it to a different computer, and resumes execution there. Work on migration of live processes in operating systems such as UNIX found a number of problems with the use of the UNIX model for migration, mainly related to *residual dependencies*. Residual dependencies arise when a process has been migrated, but needs to communicate with another process that has not, or to access files on the original host. In some cases it is possible to adapt the operating system, for instance by turning a local file system into a distributed one, but other issues such as shared memory between processes, are harder to work around while still providing the illusion of local execution (known also as a *single system image)*. The short life-times of many UNIX-processes (such as a compiler spawned from a build script) are also problematic, as a process is likely to have terminated before the migration facility has had time to decide where to migrate it.

With our work, we are attempting to build a cluster management system that places as few restrictions on users as possible, is compatible with most or all existing application software, and which provides preemption and mobility, without the problem of residual dependencies. The core of our solution is the use of virtual machines (VMs) rather than traditional processes as units of execution and migration. A VM encapsulates all program state in a single object which is easy to manage, and which may run independently of location, as long as the hosting CPU provides a compatible instruction set. Because the VM encapsulates all elements of a computation, where a process often is only a small part, the VM will typically be running for much longer, and thus be a more suitable candidate for migration. VMs can also provide performance- and name space isolation.

High performance computing clusters are powerful and often process valuable and sensitive data, and therefore they are likely to attract the attention of attackers. The attacker may be an outside entity with the simple goal of stealing access to cluster resources, or a paying customer trying to spy on other customers in order to steal business secrets. The risk of losing valuable information, and the fear of intrusions that are costly to recover from, has led to some cluster owners shutting off outside access to their resources, and thus reducing the potential for sharing and overall utilization. Our work tries to address this **security problem**, mainly by the use of compartmentalization, and by attempting to minimize the amount of privileged software (also known as the *trusted computing base (TCB)*) installed on each node in the cluster. In contrast to contemporary VM-based cluster management systems, we aim to drastically reduce the amount of permanently installed and trusted software on each computing node. We achieve this by moving functionality from the base install and into the customer-supplied and untrusted application software.

1.3 Thesis

According to the US National Information Systems Security Glossary [NSFISSI, 1997], the trusted computing base is: "The totality of protection mechanisms within a computer system, including hardware, firmware, and software, the combination of which is responsible for enforcing a security policy."

We aim to demonstrate the practicality of designing and implementing a system that allows customers to submit computational jobs in the form of self-contained virtual machines, and allows these to migrate freely between cluster nodes, without inflating the TCB.

This leads us to formulate the following thesis:

Virtual machine mobility does not have to be a feature of the virtual machine monitor, host environment, or network control plane, but can be implemented outside of the trusted computing base, using self-migration.

The purpose of this dissertation is to support the thesis, by describing a useful and powerful system built using VM technology, without having to modify the underlying VMM, and while keeping privileged software small and simple by design.

1.4 Present State

Ideally, the VMM itself only contains functionality for safe multiplexing of the CPU and hardware resources such as disk and network, providing the illusion of multiple, physical machines. Traditionally, the VMM is then augmented with a complex, network facing *control plane*, that allows the administrator to perform administrative tasks such as job instantiation and, in recent systems, job migration between physical machines.

The control plane runs with super-user privileges, and usually contains at least the following components:

1. A full TCP/IP implementation (stack), to allow remote control and act as a transport layer for VM migrations.
2. An RPC [Birrell and Nelson, 1984] mechanism for remote control by a centralized control center, or load balancing service.
3. The ability to receive incoming migrations, and to decode binary OS images for instantiation.
4. Abstractions for indirect access to paging hardware, to allow checkpointing and live migration, and page-fault logging to support a live migration algorithm.
5. Concurrency support, in the form of a thread package, to allow multiple concurrent migrations or new VM instantiations.
6. Encryption or signature checking software, to verify the validity of customer credentials.

Currently popular VMM control planes inflate the TCB with a full Linux operating system, including a TCP/IP stack, a decoder for VM images, a *shadow page table* mechanism in the VMM, and a full encryption suite, such as SSL. In total, the TCB will be increased with millions of lines of code. To be able to instantiate and destroy VMs, all or most of this code must run with super-user privileges, and this can lead to security vulnerabilities affecting the entire system.

The thesis states that it is possible to reduce this code by several factors, by moving most of the functionality out of the TCB and into unprivileged VM containers. Because we are moving functionality to higher layers, closer to the applications, we also refer to our approach as *end-to-end virtual machine mobility.*

1.5 Work Done

As part of this work we have designed and built a cluster management system called "Evil Man", a name derived[1] from the *On Demand* slogan used by IBM to describe their vision of what has now evolved into Autonomic Computing. We share their aim of making computers easier to deploy and manage, and we propose to do this by simplifying and standardizing the base software installed on each computing node. We have developed software simple and flexible enough that it may serve as a foundation for most existing software, while adding features that result in improved manageability, location independence, and isolation.

Evil Man consists of the following elements:

- A network boot-strapping protocol for virtual machines, running under the Xen [Barham et al., 2003] VMM.
- A self-migration implementation for Linux, provided as a 2000-lines source code patch.
- A small in-VM guest operating system used as the foundation for a boot-loader. The guest OS has the ability to receive executable code from the network, and to boot-strap this code into a complete guest OS, such as Linux. It can also be used for receiving incoming VM migrations.
- A trusted service that controls the instantiation of new VMs in response to incoming network commands. This service includes a HMAC signature-checking implementation, based on SHA-1 [U.S. Department of Commerce, 2002].

Due to the fast-paced nature of Xen and Linux development, a lot of work has gone into keeping our software up to date with the upstream versions of these systems. The most recent implementation is based on version 3.0.3 of Xen and version 2.6.16 of Linux.

1.6 Results

The results of this work have been encouraging. We have shown that it is possible to migrate a full virtual machine running production-level application software, with a freeze time that in most cases will be imperceptible, across a fast local area network. We have also shown how to achieve functionality equal to that of competing systems, but with only a fraction of the privileged code. Figure 1.1 shows packet latencies for two Quake 2 clients, playing on a game server in a VM that is self-migrating across a

[1]The name, Evil Man, comes from on-demand computing: if you split on-demand after the fourth letter instead of after the second then it becomes "Onde Mand" which, in Danish, means Evil Man.

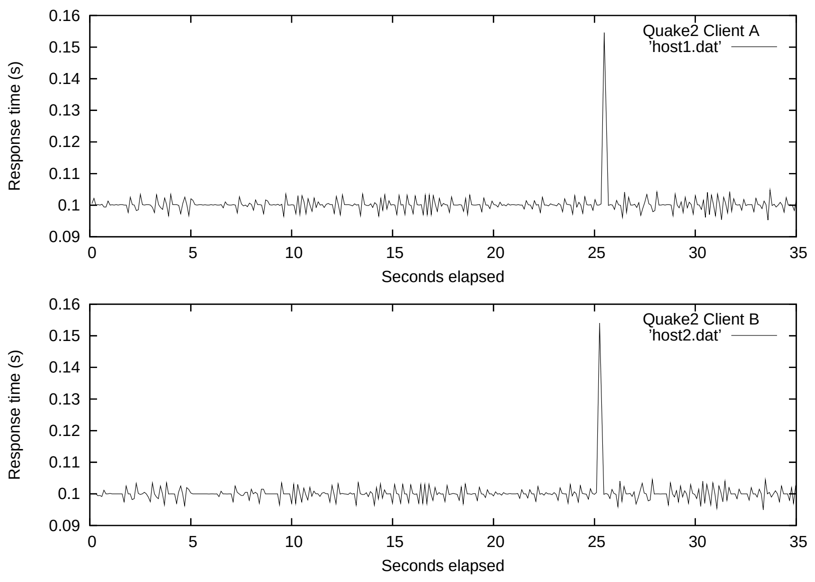


Figure 1.1: Packet latencies for a Quake 2 server with two clients.

100Mbit network. In this case, self-migration of the server VM adds only 50ms to the server response time.

1.7 Contributions

This dissertation makes the following contributions:

- The design of the self-migration algorithm.
- The implementation of self-migration in the Linux 2.4 and 2.6 operating systems.
- The design of a simple token system that lets untrusted users pay for resource access.
- The design and implementation of a minimal network control plane for on-demand cluster computing nodes, consisting of less than 400 lines of C code.
- The implementation of the Evil Man and Evil Twin system prototypes, for grid and on-demand computing.
- Experimental validation of the self-migration and control plane systems.
- The design and implementation of the "Blink" trusted display system, including a JIT compiler and an efficient interpreter for a superset of OpenGL.

- The design and implementation of the "Pulse" event notification system for simple, secure, and scalable wide-area cache invalidation.

1.8 Publications

The work described in this dissertation has resulted in the following publications:

[Hansen and Jul, 2004] Jacob G. Hansen and Eric Jul: "Self-migration of Operating Systems". In Proceedings of the 2004 ACM SIGOPS European Workshop, Leuven, Belgium, September 2004.

This workshop paper, presented at the SIGOPS European Workshop, describes both the NomadBIOS system and our first self-migration prototype.

[Hansen and Jul, 2005] Jacob Gorm Hansen, Eric Jul, "Optimizing Grid Application Setup Using Operating System Mobility". In Lecture Notes in Computer Science, Volume 3470, June 2005.

This workshop paper was presented as a poster on the European Grid Conference in 2005. It describes early self-migration performance figures, and contains the first blueprint for the Evil Man cluster management system.

[Clark et al., 2005] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield: "Live Migration of Virtual Machines". In Proceedings of the second USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA, May 2005.

This joint paper with the Xen group at the Cambridge University Computer Lab describes live migration for Xen VMs, using both our self-migration implementation and the hosted implementation developed for Xen. The paper contains benchmark results and working-set measures for a number of realistic workloads.

[Hansen et al., 2006] Jacob Gorm Hansen, Eske Christiansen, and Eric Jul: "The Laundromat Model for Autonomic Cluster Computing". In Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC), Dublin, Ireland, June 2006.

This paper describes the design and implementation of the Evil Man cluster management system, including the token system used for authentication, and performance measurements.

[Cox et al., 2006] Richard S. Cox, Jacob G. Hansen, Steven D. Gribble, and Henry M. Levy: "A Safety-Oriented Platform for Web Applications". In Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland, CA, USA.

This paper, from the University of Washington, describes the design and implementation of a Browser OS (BOS)—a sandboxing mechanism for web applications. The BOS includes the first generation of a high-performance display system for virtual machines, that later evolved into the Blink 3D display system.

[Hansen, 2007] Jacob Gorm Hansen: "Blink: Advanced Display Multiplexing for Virtualized Applications". To appear at the 17th International workshop on Network and Operating Systems Support for Digital Audio & Video (NOSSDAV), Urbana-Champaign, IL, USA, June 2007.

This paper describes Blink, a 3D display system for Xen VMs. The paper is largely equivalent to chapter 9 of this dissertation.

[Hansen et al., 2007] Jacob Gorm Hansen, Eske Christiansen, and Eric Jul: "Evil Twins: Two Models for TCB Reduction in Clusters". To appear in Operating Systems Review, July 2007, special issue on Small Kernel Systems. Edited by Michael Hohmuth.

This paper was originally intended as a straight journal-conversion of the ICAC paper above, but also contains a description and some results from more recent work on the centralized management model, which is described in chapter 8 of this dissertation.

During the course of this work, a number of supporting tools and technologies have been developed. They include the Pulse event notification system (briefly described in 8), and the EDelta algorithm for delta compression of executables. These contributions have not been described in a form suitable for publication, but some of the results from EDelta were recently published by an independent group of authors [Motta et al., 2007], at the Data Compression Conference in Snowbird, Utah.

1.9 Structure of the Dissertation

The dissertation is split into the following parts: Part one gives an overview of the history and current state of operating systems and runtime environments and provides background for the rest of the thesis. Part two describes the design and implementation of a cluster computing system sporting strong isolation, mobility, and the ability to host arbitrary software. Finally, part three discusses related work, and presents the overall conclusion. Part four contains appendices.

Part I

Background

CHAPTER 2

Virtual Machines

There is a desire to divide any system into modules, each of a size that is easy to understand and thus to maintain and reason about, and it is often beneficial to constrain the side-effects a module can have, to prevent cascading failures. Modules exist at many levels of a computing system—from procedure-blocks of instructions and objects that encapsulate code and data, over process models such as the virtual address spaces in UNIX [Ritchie and Thompson, 1974] and VMS [Levy and Lipman, 1982], and to interpreted or monitored virtual machine models, such as Java [Lindholm and Yellin, 1996], IBM VM/370 [Creasy, 1981], VMWare [Sugerman et al., 2001], or Xen [Barham et al., 2003].

2.1 Protection Kernels

One of the features we have come to expect from a modern operating system is address space protection. Early multi-tasking systems were unable to enforce any addressing limits on programs, and this meant that a stray pointer in program A could crash program B, and perhaps also the entire system. This problem can be solved in different ways, *e.g.*, in hardware using segmentation [Saltzer and Schroeder, 1974] or paging, or in software by writing all programs in a type-safe language.[1] Segmentation is a rather simple mechanism for bounds-checking of memory addresses in hardware. Memory addresses are referenced relative to a segment descriptor, specifying a base offset and a top limit of the segment. Paging is more elaborate, allowing a virtual address space to be composed out of a list of fixed-sized memory chunks, controlled by some form of dictionary data structure, mapping from virtual to physical memory addresses.

The MULTICS [Bensoussan et al., 1972] system used paging and segments to provide protection and sharing. MULTICS was an early on-line system, allowing multiple users to be logged in at the same time. Because users were often running instances of the same programs, MULTICS could share the program text between users, using a shared read-only segment.

When using paging or segmentation for protection enforcement, the operating system will need to sanity-check segment descriptors and page tables before they are installed in hardware registers. Because ordinary user programs cannot be trusted to do this

[1]See section 4.6 of chapter 4 for a brief discussion on type-safe operating systems.

correctly, the common solution is to place this functionality in a *kernel*, a trusted program with special privileges, *i.e.*, running in the processor's supervisor-mode. User programs can call into the kernel through a predefined set of entry points, but cannot access kernel data structures or bypass the kernel interface in other ways. This is enforced by executing application software in a special, restricted processor mode, known as *user-mode.* The kernel runs in the more privileged *supervisor-mode*, and is often also referred to as the *Supervisor*.

MULTICS is an early example of a *kernelized* system, where a special program is trusted and runs with supervisor-privileges, and where normal user-applications have to invoke the kernel to perform operations on shared data or peripherals. More recent kernelized systems include UNIX, Linux, and Windows NT.

One common problem with kernelized designs is the tension between the desire to include functionality, such as hierarchical file systems or device drivers, in the kernel, and the need for keeping the kernel simple and secure. In contrast to *monolithic* system that include everything in the kernel, *micro-kernel* systems such as the RC4000 Nucleus [Hansen, 1970], MACH [Rashid et al., 1989], Amoeba [Tanenbaum et al., 1990] and L4 [Härtig et al., 1997], include only the mechanisms necessary for sharing and protection in the kernel. Higher-level services such as file systems reside in ordinary user-mode processes. The main problem with this approach is the overhead of communication between the application and the many service processes.

2.2 Virtual Machines

Compared to traditional, MULTICS-derived operating systems that emphasize sharing of both resources of information, a Virtual Machine (VM) eliminates sharing, and trades utilization and coordination for isolation and scalability. The control program responsible for efficient hosting of multiple virtual machines is known as the *Virtual Machine Monitor (VMM)*. Because it executes at a privilege level higher than that of the OS supervisor, it is often also referred to as the *Hypervisor.*

Virtual machines were invented by IBM as a development tool in the 1960s, to allow two development teams to share a single computer. According to an early definition by Popek and Goldberg [1974], "a virtual machine is taken to be an efficient, isolated duplicate of the real machine", with the following three properties:

Efficiency That "innocuous" (unprivileged) instructions are executed natively by the processor.

Resource Control That a program running in a VM cannot affect its own resource allowance, without going through the VMM.

Equivalence That a program executing in a VM should perform just as it would on the real hardware, with the possible exception of timing and resource availability discrepancies.

Because of the Efficiency property, systems that emulate foreign or synthetic instruction sets (such as the Java Virtual Machine) do not fall under the original virtual machine definition. Such systems should, according to Popek and Goldberg, be referred to as *simulators*, not virtual machines.

To support the Equivalence property, the VMM must trap all instructions that read or write global machine state, and emulate their results to fool the program inside the VM into believing that it alone controls the machine. Not all processor architectures allow this to be done efficiently, *e.g.*, the Intel x86 family, before the recent addition of the VT-extensions, did not [Robin and Irvine, 2001].

In the beginning, the Resource Control property was of interest mainly to large companies, where multiple, perhaps competing or incompatible, workloads could be consolidated onto a single mainframe computer. When the Personal Computer (PC) emerged in the late 1970s and early 1980s, it did not have the capacity for running more than a single operating system, and users rarely had the need for doing so either.

Personal computers have since grown to capacities frequently exceeding the needs of a single user or application, and PC hardware is now commonly used in data centers. Though the machines may be cheap compared to mainframes, the cost of energy, connectivity, rack space, and maintenance, still leads to a non-trivial cost of per-PC ownership, and a desire to consolidate multiple workloads onto fewer machines.

Another feature of virtual machines that became important with the advent of shared-memory multi-processor machines is scalability. In a traditional operating system, the high degree of sharing between processes on different processors easily leads to lock contention, and the inability to scale beyond a handful of processors. This observation lead a group of researchers at Stanford to implement the Disco [Bugnion et al., 1997] VMM for their Flash multiprocessor system, to host multiple copies of the Irix OS. Because each OS instance was independent, there was no lock contention, and scalability was improved. The Disco system was since commercialized as "VMWare" for the x86. VMWare used dynamic binary translation to work around the limitations of the x86 platform, and allowed multiple commodity operating systems, such as Windows NT or Linux, to be hosted on a single PC.

VMWare, along with the advent of more powerful PCs, caused a renewed interest in virtual machines, and other VM systems followed. The release of the Xen VMM as open source allowed researchers the chance to experiment with VM technology, and resulted in a wealth of follow-up research on new uses of VMs.

The resurgence of VMs is not only a response to the growth in hardware capabilities, but also an attempt to solve the security problems that are increasingly threatening to destabilize or destroy many of the distributed computer systems that have become central to the efficient functioning of modern society. According to Metcalfe's law, the value of a network (such as the Internet), is proportional to the number of nodes,[2] and similar economics seem to apply to the gain of criminal activities exploiting the network [Thomas and Martin, 2006]. End-users and service providers must take measures to protect information and online assets, and VMs may provide a part of the solution. To paraphrase Creasy [1981]: "A virtual machine cannot be compromised by the operation of any other virtual machine. It provides a private, secure, and reliable computing environment for its users, distinct and isolated like today's personal computer." These isolation properties make VMs ideal containers for untrusted code, and for preventing the compromise of a single service in a VM from cascading to the entire host.

2.3 Different Types of VMs

2.3.1 System-level VMs

The term "virtual machine" has broadened over time. In the original sense, IBM developed VMs to allow two teams to share a single computer when doing low-level operating system development. Their VM closely resembled the underlying hardware, so that code developed in the VM could run in production systems and vice-versa. This work eventually lead to the IBM VM/370 [Creasy, 1981] time-sharing system. VM/370 originally hosted multiple instances of the Conversational Monitor System (CMS), but more recent versions also host more modern OSes such as UNIX. In a *system-level VM*, the input program is in machine-code format and is typically compatible with the underlying hardware. Most instructions can be safely executed directly by the CPU, with only a small subset of supervisor-level instructions requiring special treatment, often in the form of single-stepping and interpretation. The pros of the system-level approach are performance, because most instructions execute natively, and the ability to host multiple commodity *guest* operating systems on a single CPU. The cons are the lack of portability due to reliance on a certain instruction set architecture (ISA), and the inability to share fine-grained objects such as files easily between VMs.

2.3.2 Language-level VMs

In the field of programming languages, the VM term describes the interpreter and language runtime for an interpreted or Just-in-Time (JIT) compiled language, such as Java. The input to a *language-level VM* is not machine-code compatible with the underlying hardware, but frequently in the form of byte-code for an abstract machine. This approach ensures portability, because the abstract instruction set may be designed in a

[2]Metcalfe's law estimates this as n^2, later estimates use more conservative formulas such as $\frac{n(n-1)}{2}$ or $n \times log(n)$

way that provides for efficient emulation across different ISAs, but often comes at the price of a performance overhead due to interpretation or JIT compilation.

The language-VM may provide a *sandboxed* execution environment, for instance by enforcing type-safety on all memory accesses, and allows language implementors to experiment with new features, for example garbage collectors or transactional memory, without having to build new hardware. A language-VM often allows the hosted program to make OS calls, such as for file or network I/O, or other forms of fine-grained data sharing with other processes running in the same operating system.

2.3.3 Pure and Para-virtualized Virtual Machines

The original IBM VM/370 was a *pure* virtual machine, in the sense that it provided a faithful emulation of the underlying hardware, including the privileged parts of the instruction set (known as the supervisor instructions). It thus adhered to Popek and Goldberg's Equivalence property.

When a machine is shared by several processes, it is important that proper *nesting* is observed. If a process is allowed to modify global state, such as the global interrupt disable flag or the page table base register, this may affect how other processes execute, and will violate overall system integrity.

The CPU already supports the unprivileged user-mode of operation, wherein certain *privileged* instructions are considered illegal and will cause a trap. A traditional process refrains from using privileged instructions, but when running an operating system in a VM, it will attempt to use privileged instructions or access certain memory areas such as memory mapped I/O ranges. To prevent the guest VM from crashing as a result of these privileged instructions, the VMM must correctly emulate their effects on the VM.

The common way of doing this is to run the OS in user-mode and then trap and single-step through any privileged instructions. Unfortunately, some CPUs, such as the older Intel x86 models, fail to properly trap all instructions that modify or observe global state [Robin and Irvine, 2001], and even when everything is trapped correctly, such as with recent virtualization-aware CPUs that support a hypervisor-mode in hardware, excessive trapping may prove costly compared to software-only techniques [Adams and Agesen, 2006].

If the source code for the guest OS that the VMM is supposed to be hosting is available, it is tempting to relax the Equivalence property, and modify the guest OS implementation to better fit the needs of the hosting VMM. For example, an OS often needs to disable interrupts on one or more CPUs, to guarantee atomic execution of critical sections. When running inside a VM, this cannot be allowed, as otherwise the VM would be able to lock up the entire machine by turning off interrupts and going into an infinite loop. Instead, the instruction disabling interrupts should be intercepted by the VMM, and the effect emulated, *i.e.*, only interrupt delivery to the VM itself should be

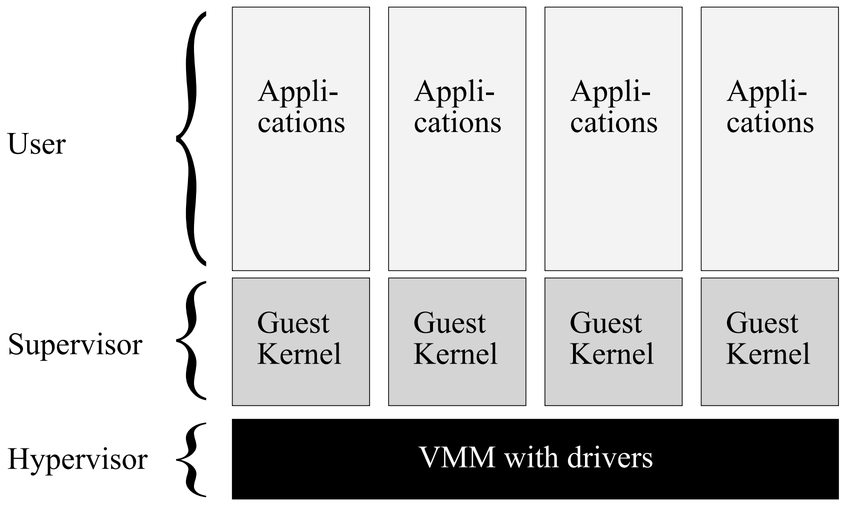


Figure 2.1: Functional layout of an event-driven, native VMM.

disabled. However, intercepting and emulating instructions may be costly, and a better alternative may be to rewrite the guest OS to call through a host API *hyper-call* or set a flag in shared memory when wanting to disable interrupts. Other functions, such as updating page tables or performing high-bandwidth I/O, may also be easier accomplished through an API rather than through emulation of hardware. The trade-off is between compatibility and performance, and classic VM papers refer to this as the pure vs. impure virtualization debate [Goldberg, 1974], whereas more recent papers refer to the use of adapted guest operating systems as *para-virtualization* [Whitaker et al., 2002]. Xen is a recent example of a para-virtualized VMM, with performance close to that of native execution [Barham et al., 2003; Matthews, 2004].

Another way to compromise on the Equivalence property, is to inform the VMM about the brand and version of the guest operating system, to allow it to optimize for the expected behavior of that particular OS. This may lead to problems if configured incorrectly, or if the guest OS is updated in a way that conflicts with the optimizations. It will also be a problem when the information is not known in advance, because the guest OS will be installed by an unprivileged user, as is the case for our network management system (see chapter 8.)

2.3.4 Native and Hosted VMMs

According to Creasy [1981], the original VM/370 hypervisor (called CP), was purely event-driven: “All work done by the machine under control of CP is the result of virtual machine action. There is no other facility, like a background job stream, to be used in place of a virtual machine.” In contrast, popular modern VMM’s such as VMWare Workstation and recent versions of Xen, are *hosted* VMMs, where a commodity OS

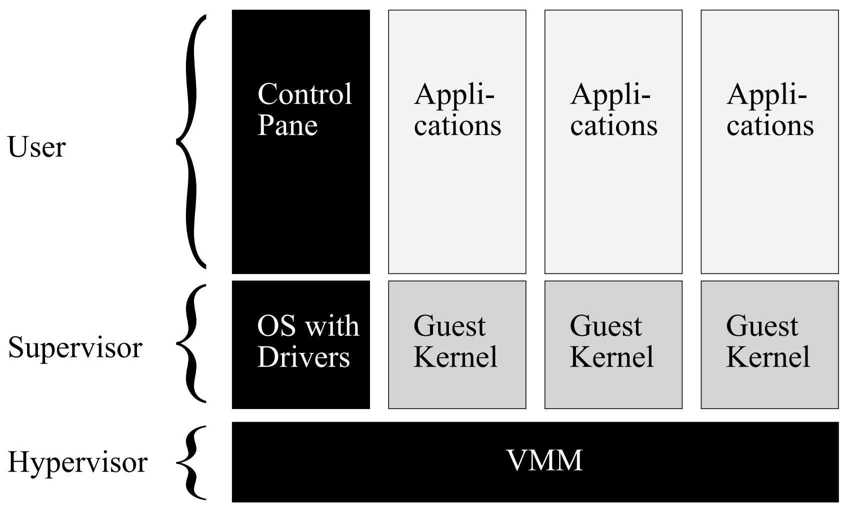


Figure 2.2: Functional layout of a hosted VMM.

runs concurrently with the VMM. VMWare ESX server, as well as the original Xen implementation described in Barham et al. [2003], are both native, event-driven VMMs. Figure 2.1 shows the functional layout of a dedicated VMM, and figure 2.2 shows a hosted VMM. The trade-off is between performance and simplicity on one side, and ease of development and deployment on the other side. A hosted VMM can utilize the drivers and other infrastructure of the hosting OS, but pays for this with reduced performance, due to context switching between VMM, host OS, and the guest VMs.

Development of a native VMM requires porting and maintenance of a set of device drivers, as well as supporting software, such as for remote management and enforcement of security policies. On the other hand, a dedicated VMM with built-in drivers does not need to context switch between guest VMs and the host OS during I/O operations, and its simpler implementation is likely to lead to enhanced security, because no vulnerabilities are inherited from the host OS.

2.4 VMs as Super-Processes

Currently popular operating systems do not allow application programmers to extend or modify the protection or concurrency primitives provided by the OS. An extensible operating system should provide the programmer a choice of protection primitives, so that, *e.g.*, one component may be protected from stray writes caused by another component crashing, without having to deal with the synchronization problems that the use of separate processes would entail.

With the introduction of the VMM as an additional layer between the hardware and the OS, the OS becomes an optional middleware rather than a necessary base technology.

This relaxes some of the constraints on the OS designer, because legacy applications can be supported in separate VMs. It also creates the option of leaving out the OS altogether, and targeting the application directly to the VM interface. This is particularly interesting for complex applications such as databases which may benefit from implementing custom mechanisms for thread-scheduling or paging.

Another area where VMs are seeing increased use is on the desktop, as containers for large, untrusted applications. Web browsers are examples of complicated applications, extended with third-party plugins, processing untrusted data. Projects such as Tahoma [Cox et al., 2006] have explored the use of VMs in such settings. We are going to discuss this subject in more detail in chapter 9.

2.5 Chapter Summary

Virtual machines allow several independent workloads to be consolidated onto a single physical machine, increasing utilization, and easing management. VMs may also be useful in other scenarios, such as on desktops, where they can be used to create an isolation barrier between mutually untrusted workloads, such as private web browsing in combination with access to a corporate network. Several types of virtual machine systems exist, with different trade-offs and different capabilities. Para-virtual VMs host adapted guest OSes with close to native performance, whereas pure VMs host any guest OS that is compatible with the underlying hardware, often with a higher overhead. A hosted VMM relies on a commodity operating system for hardware access, whereas a dedicated VMM includes its own set of drivers.

CHAPTER 3

Checkpoints and Mobility

In this chapter we discuss different mobility systems, from fine-grained *object mobility* systems and programming languages, over operating systems that support *process migration*, to coarse-grained *virtual machine migration* systems. Common to all of these systems is that they use some form of checkpointing mechanism in order to consistently migrate program state between physical machines.

A *checkpoint* is a snapshot of all application state, potentially with the exception of soft-state that can be recreated on demand. A checkpoint c of a process p is *consistent* at the time t_c, if and only if any input event e to p happening before t_c is recorded in c, and no event happening after t is recorded in c.

It is often desirable to move long-running computations between processors, or, in a cluster system, between different hosts altogether, and a checkpointing facility is often the main ingredient of such *mobility* systems. A checkpoint is captured at the source host, transmitted to the destination host, and resumed there.

Checkpoints can be taken of individual processes, or of an entire operating system. A process-level checkpoint will often fail to capture all relevant state, such as state in open files or shared memory segments, whereas a system-wide checkpoint will be able to capture all local state. Distributed state is less trivial to capture, but the failure modes that can arise from the loss of distributed state is better understood than for a single-process checkpoint. Whereas few application programs can handle the sudden loss of an open file or state kept in a shared memory segment, most OS network stacks will be able to cope with the loss of a few network packets.[1]

Even though complete checkpoints require more storage than application-specific data formats, they can still be useful. The main advantages are that a system-wide checkpoint is able to capture the state of multiple independent threads and processes, and that checkpointing can be provided as a global and application-transparent feature. Checkpoints are useful in coarse-grained mobility systems, where a group of processes are moved as a whole.

One application of checkpointing is "hibernation"—the ability to store a checkpoint of a running system on stable storage. Many laptop computers have a "suspend-to-disk" feature, where either the BIOS or the operating system has the ability to checkpoint itself to disk or nonvolatile memory, to allow the user to continue work at some later

[1]Here, we are concerned mainly with the use of checkpoints for mobility, and consider the wider issues of distributed checkpoint correctness to be out of the scope of this work.

point in time. This functionality is weaker than what a persistent system provides, but because the computer is turned off immediately after the checkpoint, also simpler to implement. All relevant in-memory state becomes part of the checkpoint, and because state on-disk state remains constant as long as the system is off, it will remain checkpoint-consistent. Because the liveness of the system is no longer a concern, most parts of it can be suspended during the checkpoint, simplifying the checkpoint implementation.

3.1 Manual and Seamless Mobility

Mobility systems aim to increase the flexibility of computing installations and to reduce the need for manual configuration. When data, objects, processes, or virtual machines, become mobile, it is possible to grow or shrink the installation simply by adding or removing hardware.

Though not many advanced mobility systems have been successful and seen widespread use, simple forms of *manual mobility* are present everywhere—from the simple act of moving a hard-drive full of system binaries to a new machine, to a cellphone downloading and running Java applets off a wireless connection. Naturally we do not consider these scenarios as genuine mobility systems, and the main reason is that they involve explicit shutdown, saving, or restoration stages. In other words, they are not examples of *seamless mobility* systems. A seamless mobility system is one in which mobility is transparent, whether to a user, to objects, or to a set of processes, and where no explicit steps are taken to externalize program state.

3.2 Fine- and Coarse-Grained Mobility Models

As described, mobility can be achieved at several levels of granularity, and both manually and transparently. In this section we describe the granularity levels, and some of the more well-known implementations.

3.2.1 Data and Object Mobility

A simple yet limited way of achieving mobility is to rely on each application correctly externalizing its state to a form from which the state of the program may later be recreated. *Data mobility* is achieved by moving the externalized data between machines. If the externalized state utilizes an efficient format, data mobility is also likely to incur lower transmission costs than full checkpoint mobility. In the many cases where data mobility is possible, *e.g.*, when the mobile program expects and supports it, data mobility is therefore likely to be more effective.

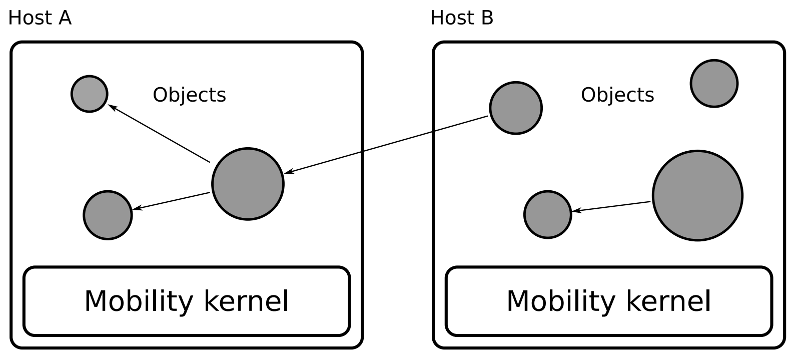


Figure 3.1: An object mobility system, with objects shown as circles and dependencies as arrows. Fine-grained objects can be moved between hosts, by the mobility kernel. Communication between the objects is transparently forwarded over the network.

The next step up from data mobility is *object mobility*. In programming language terms, an *object* encapsulates code and data as a single unit, and object mobility can be achieved by moving serialized objects between hosts. If all inter-object communication happens through object method invocations (calls to procedures defined in the scope of an object), *remote method invocation (RMI)* may be used for transparently forwarding calls to a remote object. In RMI and object mobility systems, a global registry is often used for keeping track of object locations, or *forwarding addresses* are installed after to help locate moved objects. If an object is *live*, *i.e.*, one or more threads are executing inside the object, and thus keep parts of the object state in CPU registers or the stack, mobility becomes more complicated, as this state must be written back to the object activation record before the object can be safely moved to another host. Some systems wait for object invocations to reach certain *safe points* of execution, before mobility is allowed. RMI and object mobility systems are often implemented as part of programming language runtimes, *e.g.*, the Eden [Black, 1985] and Emerald [Jul et al., 1988] kernels or the Java Virtual Machine [Lindholm and Yellin, 1996]. Thus they function independently of the underlying operating system. Figure 3.1 shows a small number of objects distributed across two hosts, with object mobility provided as a transparent service by a mobility kernel.

3.2.2 Process Migration

Object mobility is fine-grained. A whole program may contain hundreds or thousands of objects, and each of these can roam freely between hosts. At the other end of the spectrum we find more coarse-grained techniques, namely process migration and virtual machine migration.

Like object mobility, process migration moves program state and data together as a single unit. Process migration systems view each process as an opaque object, and are often implemented as a feature of the underlying operating system. The advantages

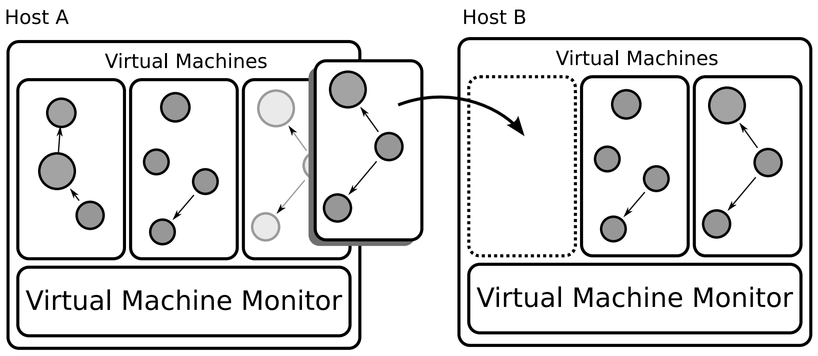


Figure 3.2: A virtual machine mobility system. Coarse-grained VMs can be moved between hosts All object and process dependencies are captured as part of the VM checkpoint, and there is no need to forward communication.

of this approach is that any process may be a candidate for migration, independently of implementation language, and that existing operating system mechanisms may be used for safely preempting and migrating processes, without having to worry about critical sections or safe points. Migration thus happens *transparently* to the process.

In contrast to object systems, the interfaces that processes communicate through are more complex, and this can cause problems or prevent migration. For example, UNIX [Ritchie and Thompson, 1974] single-copy semantics stipulate that all writes to a file are immediately visible to readers of that file, and this causes problems when some readers have been transparently migrated to a remote host. Because migration is transparent, processes will not expect file system semantics to change to a weaker form of consistency, causing process failure or incorrect execution. Some types of processes may depend on shared memory for communication, causing further complications.

Process migration systems suffer from the problem of *residual dependencies*. This problem occurs when a process leaves behind open files or other state in the operating system on the originating host. Both systems attempt to solve this problem by leaving a proxy processes on the originating host, handling access to local resources on behalf of the migrated process. This solution is problematic for two reasons; performance and stability. If all access to a resource has to go via the network, execution may be slowed, and by relying on resources on the originating host, the vulnerability of the migrated process to a machine crash is increased, because the process will fail if just one of the two involved hosts crashes.

Examples of operating systems supporting process migration include Distributed V [Theimer et al., 1985], Sprite [Douglis and Ousterhout, 1991] and MOSIX [Barak and La'adan, 1998]. We will discuss these as related work in chapter 10.

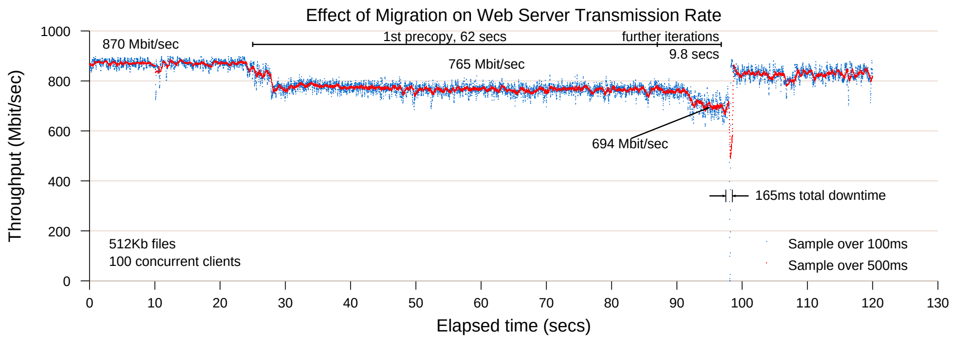


Figure 3.3: Throughput profile of a migrating VM running SpecWeb99 (from Clark et al. [2005].)

3.2.3 Virtual Machine Migration

When VMs are used for consolidation of server workloads, hardware utilization is increased, but so is the impact of server downtime, because a larger number of applications rely on the availability of fewer physical servers. For planned or predictable downtime, this problem can be entirely solved by the use of virtual machine migration, where workloads are relocated to other hosts before the original host is shut down. One disadvantage, compared to finer-grained migration mechanisms, is that VMs can be large, especially if disk state has to be migrated along with the VM.

As shown in figure 3.2, virtual machine migration is the most coarse-grained type of mobility. As described in the previous chapter, a system-level VM is one that can run a commodity operating system, and thus is able to host unmodified software written for that OS. If the entire VM is moved to a new host, all applications running in the VM will move as well. The file system and shared memory problems plaguing process migration systems are therefore solved, as are indeed many residual dependency problems. Two residual dependency problems that must still be addressed are access to disk storage, and the migration of network connections. In the context of a local area network, both of these problems can be solved by utilizing existing mechanisms, *e.g.*, network-attached storage, and the address resolution protocol used when running IP over an Ethernet. We discuss storage for VMs in chapter 7, and networking in chapter 7.

3.3 Migration Freeze Time

Systems supporting transparent mobility often strive to make a location change transparent not only to the program being migrated, but also to external observers. Software which is not purely batch-oriented, *e.g.*, software interacting with human users, or software which is part of a distributed system, needs to respond to requests in a timely manner, as external peers often interpret a prolonged silence as a sign of a crash or disconnect. Thus, a system which communicates with the outside world is often subject to soft real-time demands, and this aspect needs to be taken into account by mobility systems. Simple “stop the world” schemes for obtaining and migrating system checkpoints are therefore inadequate for many uses, and iterative algorithms with better real-time characteristics are often preferable, because they shorten the “freeze time” during which the migrated program is unresponsive.

Below, we describe three popular approaches to migration, and the measures that they take to reduce migration freeze times:

Stop-and-copy The task is stopped, and memory as well as task state information is transferred to the new host, where the state is restored and the task resumed. This scheme is the simplest scheme, and transfers all data only once. The task is unresponsive during the entire copy.

Lazy copy In lazy-copy migration [Zayas, 1987], the migrating task is stopped and kernel and CPU task state is transferred to the new host, where the task is resumed immediately, inside an empty address space. When the task page-faults, the faulting page is fetched from the old host before the task is allowed to continue. The task is only unresponsive for the short amount of time taken to transfer task state to the new host, but performance of the task running at the new host suffers initially, since every page-fault must be resolved across the network. Pages that are not referenced can optionally be transmitted in the background, to avoid leaving a residual dependency trail on the original machine. One problem with lazy migration is that a failure of the network or the source or destination host during migration, will result in irrecoverable failure of the migrating task, because no hosts possesses a current and complete version of the entire task state.

Pre-copy In pre-copy migration [Theimer et al., 1985], the task is left running at the source machine and all memory owned by the task is mapped read-only and copied to the new host. When the still running task attempts to modify one of its pages, a page fault is raised, and the page marked as dirty. The page is then mapped read-write to the task, which continues running.

After the initial transfer, a subset of the pages will be dirty, and these are again mapped read-only and copied to the new host. This goes on until the dirty subset is sufficiently small, after which the task is suspended at the originating host, the remaining dirty pages copied across, and the task is resumed on the new host.

The downtime of the task is reduced to the time taken to copy the last set of dirty pages to the new host, but a number of pages will be copied more than once. Figure 3.3 shows how a large VM running the SPECweb99 benchmark can be migrated with very little downtime, by use of the pre-copy algorithm.

Pre-copy migration leaves no residual dependencies on the originating host, and

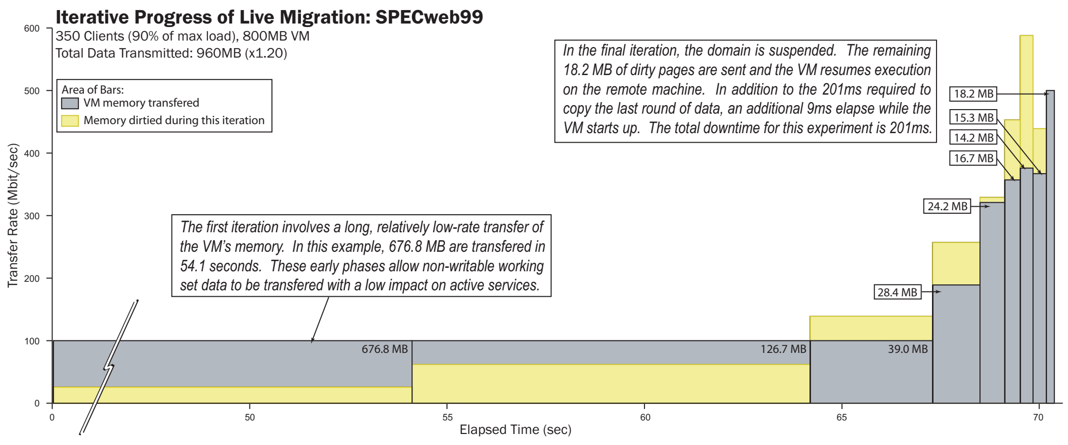


Figure 3.4: Timeline for pre-copy migration of a VM running SPECweb (from Clark et al. [2005].)

does not suffer from the increased risk of failure present in lazy copy. Should the network or the destination host fail during migration, the source host can re-resume control. Figure 3.4 shows an annotated timeline for a pre-copy migration of the VM running SPECweb99.

3.4 Chapter Summary

The use of a transparent checkpointing facility allows for migration (mobility) of running programs across distinct physical hosts. Migration can be implemented at different levels, *e.g.*, by language runtimes, by operating systems, or for entire operating systems running in virtual machines. Mobility can be useful for load-balancing, for moving code closer to data, and for improving system availability. Most mobility systems suffer from the residual dependency problem, that can prevent an object or process from migrating, or that can be detrimental to performance or availability of a migrated process, if not handled well. VM migration systems have the advantage that they encapsulate all dependencies inside the migrating VM, but VMs can be large and take longer to migrate.

We will continue the discussion on virtual machine mobility in chapter 5, where we are also going to describe our technique for operating system self-migration.

CHAPTER 4

Device Drivers

Device drivers pose a special challenge for OS designers, because they need unrestricted hardware access, while at the same time being user-installable and upgradeable. Virtual machine monitors frequently provide idealized abstractions through which guest operating systems can get hardware access, but at the lowest level the VMM still must implement a set of device drivers for the actual hardware present in the machine.

Furthermore, applications depend on drivers for correct execution, and drivers may contain state that is hard or impossible to extract if the driver needs restarting, or the application wishes to migrate to alternative hardware. Our work has not dealt with device drivers primarily, but how device drivers are implemented and where in a system they are located, has a profound effect on how the remainder of the system can be implemented. The purpose of this chapter is to provide the reader with background for the discussion and design choices with regards to device access presented in subsequent chapters.

4.1 Design Space

Software that is unable to communicate with the outside world is of limited use. A system for on-demand computing, such as the one we are trying to build, needs to provide applications with multiplexed peripheral access, *e.g.*, to the network for communication with the end user or with jobs on other nodes, and to disk storage for saving interim results or staging input data for future use. The device driver interface that the base software provides influences how application software is written, and should be defined in a way that does not unduly limit future development or access to new and smarter peripherals. The actual placement of device drivers influences the amount of privileged and trusted code, and may also impact performance to a great extent.

The design space is defined by parameters such as whether to expose a high-level (*e.g.*, the UNIX socket API) interface or a low-level one (*e.g.*, raw Ethernet frames). At the lowest level, it is also possible to allow applications to access peripheral hardware directly, in the style of an Exokernel [Kaashoek et al., 1997]. High-level models are often easier to multiplex, because the multiplexing layer has a better understanding of the workload and more freedom to adapt the workload (for example by reordering disk requests or batching network packets) to the current situation. On the other hand, low-level models provide the application with more flexibility, such as the ability to upgrade a transport protocol without changing the underlying layers.

4.2 In-Kernel Device Drivers

In contrast to processes that execute within a protection boundary, device drivers in the majority of deployed systems are frequently allowed unhindered access to the OS address space and resources. There are a number of motivations for such a tightly coupled design:

Performance During execution, the driver needs to interact with the OS to allocate resources, to influence scheduling, or to communicate with shared subsystems, such as the TCP/IP stack of file system. This is simpler to achieve when the driver resides within and has full access to the OS address space.

Application memory access The majority of devices employ DMA to operate asynchronously from the CPU(s). If, say, an application queues a write request for the disk driver, the driver will need to *pin* the data for the request in system memory, until the request has been completed. Pinning of memory is simpler to achieve with tight coupling between driver and OS, and essential to performance.

No driver API Defining a programming interface (or API) for driver access is hard, because future hardware developments cannot always be predicted. For example, a driver API for an Ethernet controller designed 10 years ago would include the ability to send and listen for Ethernet frames, but would likely be missing interfaces to TCP-offload mechanisms currently present on many controllers. A driver residing with the OS will be able to access arbitrary OS interfaces, in this case TCP/IP stack internals, and as a result novel hardware features will be easier to support.

The DMA problem The final point is that few systems at this point provide a hardware indirection layer (or IO-MMU) for device direct memory access. In other words, there is no way a malicious or badly programmed device may be prevented from accessing arbitrary memory-regions. Thus, with current hardware, most driver encapsulation schemes may still fail in some conditions, and are vulnerable to malicious drivers. It is thus arguable whether the overhead introduced by device driver encapsulation techniques is at all justifiable in systems without an IO-MMU. Fortunately, IO-MMUs are on the horizon, *e.g.*, AMD's Pacifica hardware contains a Device Exclusion Vector, a bitmap that decides which memory pages can be accessed by devices.

This tight coupling is of course problematic if the driver contains bugs, or if the driver installation API is used as a vehicle for installing trojans or spyware into an otherwise trusted kernel. Various micro-kernel systems have advocated a loose coupling model, with device drivers being treated more like processes. One of the earliest designs with this characteristic was the RC4000 *nucleus* described by Hansen [1970], where special internal processes were used for dealing with access to peripheral hardware. The RC4000 did not sport hardware memory protection, and thus protection-wise processes and drivers were equivalent. More recent designs, such as the L4 microkernel, permitted processes (*tasks* in L4 terminology) to receive hardware interrupts formatted as IPC messages, and access to memory mapped I/O registers and I/O port space could be controlled using virtual memory access *grants*. The Nooks [Swift et al., 2005] system attempted to retrofit driver isolation into an existing Linux operating system, by allowing device drivers to execute in the normal virtual address ranges, but with access permissions set more restrictively than normal. Neither of these systems attempted to deal with the DMA problem, and while still able to contain many accidental types of faults, were not able to withstand attacks from directly malicious device drivers. Furthermore, the cost of making changes to virtual address spaces by context switching often added quite large execution overheads, because drivers, which normally run for very short times before returning control to the operating system of controlling user application, often are unable to amortize large switching costs.

4.3 Use of VMs for driver isolation

Similar to microkernels, it has been suggested [LeVasseur et al., 2004; Fraser et al., 2004] that VMs could provide a solution to the problem of device driver isolation. Indeed, by providing idealized, abstracts APIs to guest VMs, VMMs are often used as compatibility layers between experimental or legacy operating systems and hardware for which these possess no driver implementations.

This approach also has downsides. VMs containing full operating systems are much more coarse-grained abstractions than traditional driver modules, so there is a large memory overhead. Because the driver VM will be a black box, the user may find it harder to configure and troubleshoot than a traditional system. Finally, this approach only solves the problem for classes of hardware for which an emulation protocol has been defined. In a traditional driver model, support for whole new classes of hardware can be added directly, whereas in the driver-VM model a communications protocol for the new class must be added to both the application and the driver VM.

4.3.1 Context Switching Overhead

On architectures supporting segments, the cost of address space switches can be reduced by using segmentation instead of paging hardware for isolation. When switching between virtual address space, the contents of the TLB need to be flushed, and the performance degradation resulting from the subsequent slew of TLB misses is a larger factor in the total cost of a context switch. If the virtual address spaces are small enough to make several of them fit inside the maximum addressable range, another

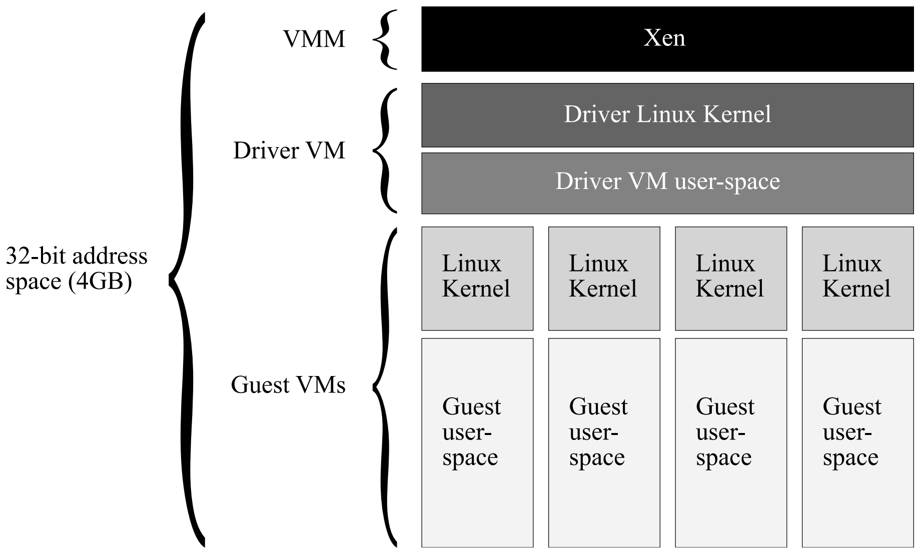


Figure 4.1: Virtual address space layout when co-locating driver and guest VMs.

option is to enforce isolation with segments. This approach was suggested by Liedtke for L4 [Liedtke, 1995]. The problem with the approach is that on 32-bit platforms, virtual address space is relatively scarce, and that some of the never 64-bit platforms (such as early versions of the AMD Opteron) have left out segmentation support. Due to initial performance problems with the Xen 2.0 "next generation I/O model" [Fraser et al., 2004], with drivers in a dedicated VM, we carried out an experiment to see if the use of segmentation would improve I/O performance. The hypothesis behind this experiment was that by co-locating the driver-hosting VM with the guest VM in a single address space, TLB flushes could be avoided, and performance would increase. The goal was to improve TCP bandwidth between a guest VM and an external machine, over a gigabit Ethernet link.

This change required modifications to the Xen VMM, to the driver and guest Linux kernels, and to the user-space applications running in the driver VM. Specifically:

- When performing a VM "world switch", Xen would check if the old and new VM would both fit in the virtual address space at the same time, and in that case skip the TLB flush.
- The driver VM was relinked to fit into a very small address space, just below the Xen VMM at the top of memory.
- *All* user-space applications running in the driver VM had to be relinked to run in a small address space, above guest kernels, but below driver VMs. In practice this was achieved by using the BusyBox[1] multi-call binary to provide most of the user-space functionality for the driver VM.
- The guest Linux kernel was modified to use a slightly smaller portion of virtual

[1]http://www.busybox.net/

memory, to make place for the driver VM and its user-space.

- The user-space applications running in the guest VMs were not modified. The system was thus still able to host unmodified guest applications.

Figure 4.1 shows the virtual address space layout of the modified Xen system. With these changes, we measured an 8% improvement in TCP bandwidth between a guest VM and a remote host, with throughput increased from 574MBits/s to 620MBits/s, using the `iperf` bandwidth measurement tool. We conjecture that real-world applications that overlap computation and communication, would see larger performance improvements. On 32-bit platforms, this technique is applicable only when driver VM applications can be relinked and have very modest virtual address space requirements. One such scenario is our Evil Man on-demand computing platform, described in section 8.2. Lack of time has so far prevented us from exploring the use of small address spaces any further.

4.4 Restarting Device Drivers

In traditional systems, drivers are not allowed to fail. Failure of a driver often leads to corruption of memory and CPU state, and in most cases the only way out is a system-wide reset. If drivers are encapsulated and allowed to fail, the consequences of failure must be taken into consideration. The Nooks project implemented a best-effort safety mechanism for drivers, but though the system is able to recover from many types of driver failure, many applications relying on correct driver functioning are not. The problem is twofold; drivers may fail in byzantine ways and feed invalid data to applications, and even if drivers are fail-stop, they are likely to contain state that must be recreated after a driver restart.

Nooks aims to be a transparent system, so that neither drivers nor applications have to be modified, and the proposed solution [Swift et al., 2004] is the insertion of a *shadow driver* between the application and the real driver. The shadow driver tracks state changing commands, so that they can be replayed to the real driver if it ever crashes and needs to restart. The problem of byzantine driver failure is not addressed by Nooks, but is nevertheless quite interesting and challenging. Detection of a byzantine driver may be non-trivial if the driver never fail-stops, but perhaps knowledge of the hardware class can be exploited, *e.g.*, an Ethernet driver should return frames in a certain format and with a valid checksum, and most network applications are programmed to survive “garbage-in”. The situation is worse for hardware such as disk drives, that are expected to behave predictably and return previously stored data unscathed. Many applications will likely crash on byzantine input files, and the only transparent solution to these problems would be the use of a lightweight checkpointing mechanism, to allow the application to roll back to safe state after the failure of a driver.

4.5 External Device State

The problem of restoring or retrieving external state in drivers or in the peripheral hardware itself, is not only related to driver crash recovery. External state also poses a big challenge to systems attempting to implement mobility, *e.g.*, process and VM migration systems.

In a VM system supporting mobility, is it important that all state relevant to the functioning of a VM either resides within the VM, or is simple to extract from the VMM or host environment. Ideally, the host/VMM should keep no state at all, as this will allow it to be policy-free with regards to the amount of state it is possible to store outside the VM. Naturally, some if this state is trusted, and cannot be safely stored in the VM without additional validity-checking being performed before it is used. This would be the case for a permission table describing areas on disks to which the VM has access, or the integer value describing the VM's memory allowance, and for the list of pages the VM is allowed to map. One possible solution would be to introduce a *caching* model, where all state in kept in the VM is *sealed* using encryption, and with the VMM caching relevant parts of this information after verifying that is has not been tampered with.

As part of this work, we implemented a 3D graphics display system, called Blink, for VMs. Blink is an example of how a driver layer can be constructed to be restartable, and to support migration and checkpointing of client applications. Blink is described in chapter 9.

4.6 Type-safe Languages

Device driver safety may also be addressed by writing drivers in type-safe languages. Safe-language kernel extensions have been proposed, e.g. the *Device Monitors* of Concurrent Pascal [Ravn, 1980], and in the SPIN [Bershad et al., 1995] and Exokernel [Kaashoek et al., 1997] operating systems. In Exokernel, applications would download code such as packet filters into the kernel, where it would be compiled into native code and check for misbehavior before letting it execute. In SPIN, and in more recent systems such as Singularity [Hunt and Larus, 2007], the compiler is trusted to produce code that abides by the type-safety rules of the implementation language. Provided that the compiler is correctly implemented, safe-language extensions can safely execute inside the OS kernel. So far, driver developers seem to prefer unsafe language such as C or C++ for driver development, and the type-safety is only a partial solution, as it does not address the DMA problem described above. Finally, there is a tension between creating a trustworthy compiler, *i.e.*, one that is simple enough that it can be thoroughly audited, and one that produces well-optimized code.

4.7 Software Fault Isolation

For device drivers that cannot be rewritten in type-safe languages, an alternative is to use a Software Fault Isolation (SFI) [Wahbe et al., 1993] or *Sandboxing* technique. Such techniques rely on a combination of software and hardware protection, but the core idea is the use of *Inlined Reference Monitors (IRM)* [Erlingsson, 2004] embedded in the program instruction stream. The IRM-code monitors program state at run-time, and throws a software exception if the security policy is violated. IRMs are typically installed during compilation, and their presence is verified before the program runs. In contrast to type-safe languages, this division of concerns removes the necessity for trusting the compiler, and only requires that the verifier is trusted. Early SFI systems made very strong assumptions about the underlying instruction set architecture (ISA), *e.g.*, about the alignment of instructions, and as such were only applicable to certain RISC architectures. Also, they were mostly targeted at user-space extensions, and could not always prevent sandboxed code from getting the CPU into a irrecoverable state by use of certain privileged instructions.

The recent XFI project [Erlingsson et al., 2006] extends SFI to also work on CISC platforms. XFI addresses the complexities of the x86 ISA, *e.g.*, the ability to jump to the middle of an instruction, by only allowing a subset of the instruction set to pass the verifier, and by enforcing *control flow integrity (CFI)*. CFI enforcement means that all computed jumps (*e.g.*, return instructions and jumps through pointer tables) are restricted to a predetermined set of addresses. With the CFI guarantee in place, it is possible to reason about the possible flows through a program, and thus to verify the presence of an IRM policy.

4.8 Chapter Summary

There are still many open problems relating to device drivers. Existing systems trade driver safety for performance and ease of development, and device drivers are a major source of system instability. Attempts have been made to improve the situation, hardware protection techniques, *e.g.*, micro-kernels and Nooks, and through software-enforced isolation. Commodity systems do not enforce addressing restrictions on device-DMA, limiting the effectiveness of the described techniques. Finally, if applications are to survive a driver crash, the OS or driver protection mechanism must have a way of recreating lost hardware state on driver reinitialization.

We have explored the use of VMs and self-checkpointing in a desktop system, and designed and implemented a high-performance display abstraction to support VM applications. This work is described in chapter 9, as well as in Cox et al. [2006] and in Hansen [2007].

Part II

Design and Implementation

CHAPTER 5

Self-Migration

This chapter describes the self-migration mechanism, the design and implementation of which is the main contribution of our work. Based on our experiences with the NomadBIOS "hosted migration" system, we found that it was possible to migrate running operating system VMs across a network, with very short freeze times, but that adding this functionality to the VMM inflated the size of the trusted computing base. These observations led us to design the self-migration algorithm, which has comparable performance, but does not inflate the trusted computing base.

5.1 Hosted Migration

Our first implementation of *hosted* VM migration was the NomadBIOS system [Hansen and Henriksen, 2002]. NomadBIOS was a prototype host-environment for running several adapted Linux-instances concurrently, and with the ability to migrate these between hosts without disrupting service (now commonly referred to as "live migration".) NomadBIOS ran on top of the L4 microkernel [Härtig et al., 1997], multiplexing physical resources for a number of concurrently running guest operating system instances. It contained a TCP/IP stack for accepting incoming migrations, and for migrating guest operating systems elsewhere. NomadBIOS ran incoming guest operating systems as new L4 *tasks* inside their own address spaces, and provided them with backing memory and filtered Ethernet access. An adapted version of L4Linux 2.2, called NomadLinux, binarily compatible with native Linux, was used as guest OS.

All guest memory was paged by NomadBIOS, so it was simple to snapshot the memory state of a guest OS and migrate it to another host, though quite a lot of cooperation from the guest was needed when extracting all thread control block state from the underlying L4 microkernel. To reduce downtime, NomadBIOS used pre-copy migration, as described in chapter 3, keeping the guest OS running at the originating host while migrating, tracking changes to its address space and sending updates containing the changes to the original image over a number of iterations. The size of the updates would typically shrink down to a hundred kilobytes or less. As a further optimization, a gratuitous ARP packet was then broadcast to the local Ethernet, to inform local peers about the move to a new interface on a new host.

NomadLinux running under NomadBIOS was benchmarked against Linux 2.2 running under VMWare Workstation 3.2 and against a native Linux 2.2 kernel where relevant. Performance was generally on par with VMWare for CPU-bound tasks, and scalability when hosting multiple guests markedly better. When compared to native Linux, the performance was equal to that of L4Linux, a 5-10% slowdown for practical applications. Migration of an OS with multiple active processes, over a 100Mb/s network, typically incurred a downtime less than one tenth of a second.

Similar results have since been achieved in both research and commercial systems. NomadBIOS inspired the live VM migration facility present in Xen [Clark et al., 2005], and a similar facility is present in the commercial VMWare ESX Server product [Nelson et al., 2005]. Compared to both NomadBIOS and our more recent work, the primary advantage of these systems is the ability to migrate unmodified guest operating systems, such as Microsoft Windows.

5.1.1 Problems with Hosted Migration

As described above, hosted migration requires extensions to the VMM and other parts of the trusted software installation on the machine. Whether or not this is a concern depends on the way the system is deployed and used. For example, VMWare recommends using a separate, trusted network for migrations, and naturally this reduces the risk of an attack on the migration service. Also, the ability to relocate a VM without its knowledge or cooperation may be practical in some scenarios.

The focus of our work is grid or utility computing, and often we cannot trust neither the network nor the applications we are hosting on behalf of customers. In addition, we should try to stay as flexible as possible, and to refrain from providing policies where mechanisms would suffice. We therefore try to avoid implementing functionality in the trusted parts of the system, where its presence may be exploited by an attacker, and where it can only be changed through a system-wide software upgrade.

5.2 Self-Migration

The experiences from developing NomadBIOS led to the design of a new system, based on the self-migration algorithm.

The self-migration implementation is based on the Xen [Barham et al., 2003] virtual machine monitor (or hypervisor) which divides a commodity Intel PC into several virtual domains, while maintaining near-native performance. Xen originally lacked comfortable abstractions common to other microkernels, such as the paging-via-IPC mechanism and recursive address spaces of L4, and did not provide any clean way of letting a guest OS delegate paging responsibilities to an external domain. However, this is required when implementing host-driven live-migration as otherwise there is no way of tracking changed pages from the outside. While the Xen team was working to add such functionality, we decided to pursue another approach: In line with the end-to-end argument [Saltzer et al., 1984], we proposed to perform the guest OS migration entirely without hypervisor involvement, by letting the guest OS migrate itself. Compared to hosted migration, this self-migration approach has a number of advantages:

Security By placing migration functionality within the unprivileged guest OS, the footprint and the chance of programming errors of the trusted computing base is reduced. For instance, the guest will use its own TCP stack, and the trusted base needs only provide a filtered network abstraction.

Accounting and performance The network and CPU cost of performing the migration is attributed to the guest OS, rather than to the host environment, which simplifies resource accounting. This has the added benefit of motivating the guest to aid migration, by not scheduling uncooperative (for instance heavily pagefaulting) processes, or by flushing buffer caches prior to migration.

Flexibility By implementing migration inside the guest OS, the choices of network protocols and security features are ultimately left to whoever configures the guest OS instance, removing the need for a network-wide standard, although a common bootstrapping protocol will have to be agreed upon.

Portability Because migration happens without hypervisor involvement, this approach is less dependent on the semantics of the hypervisor, and can be ported across different hypervisors and microkernels, and perhaps even to the bare hardware.

Support for Direct I/O When virtualizing the CPU, current VMMs come close to having native performance. The performance of network and disk I/O is more of a problem, especially in hosted VMMs such as Xen and VMWare Workstation, where a context switch is required when servicing interrupts. One possible solution is the use of special hardware that can be safely controlled directly by the unprivileged VM, without involving the VMM or a host operating system. Unfortunately this model works badly with hosted migration, because the state of the guest VM may change without the VMMs knowledge, through direct device DMA. Self-migration, on the other hand, is not in conflict with direct I/O mechanisms.

The main drawbacks of self-migration are the following:

Implementation Effort for Each Guest OS Self-migration has to be re-implemented for each type of guest OS. While implementing self-migration in Linux proved relatively simple, this may not generalize to other systems. Also, developments in the guest OS may result in incompatibilities, and will require ongoing maintenance.

Only Works for Para-virtualized Guests Self-migration requires that each guest OS is extended with new functionality. In cases where this is not possible, such as when not having access to the source code of the guest OS, self-migration will not work. In some cases, it may be possible to extend the OS by installing a self-migration driver as a binary kernel module.

Because we are targeting on-demand, high performance computing, we have mostly been focusing on open source operating systems such as Linux. Linux is popular in these settings because of its low cost, and because it can be customized easily. When running each workload in a separate VM, the licensing costs of a commercial OS could also become a serious issue. Self-migration may be less applicable in other situations, where the ability to host proprietary operating systems such Microsoft Windows is of higher importance.

5.2.1 Self-migration Algorithm

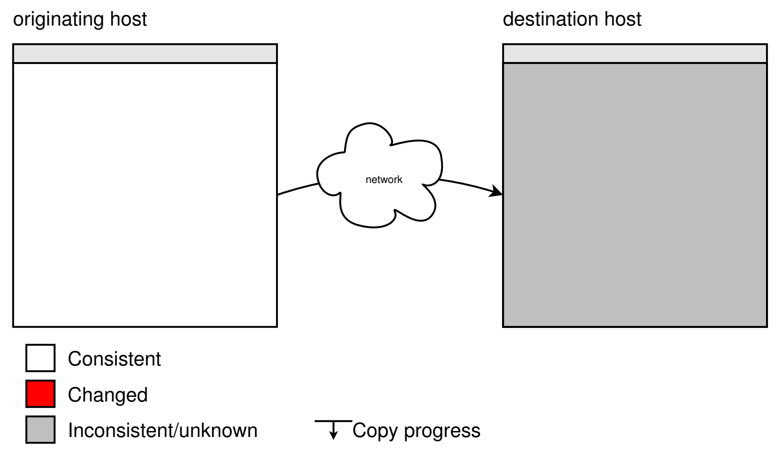
Our self-migrating OS is based on the existing Xen port of Linux, known as XenLinux, which we have extended with functionality allowing it to transfer a copy of its entire state to another physical host, while retaining normal operations. The technique used is a synthesis of the pre-copy migration algorithm and of the technique used for checkpointing in persistent systems.

The main drawback of self-migration is that it needs to be re-implemented for each type of guest OS. While implementing self-migration in Linux proved relatively simple, this may not generalize to other systems. Our self-migrating OS is based on the existing Xen port of Linux, known as XenLinux, which we have extended with functionality allowing it to transfer a copy of its entire state to another physical host, while retaining normal operations. The technique used is a synthesis of the pre-copy migration algorithm and of the technique used for checkpointing in persistent systems.

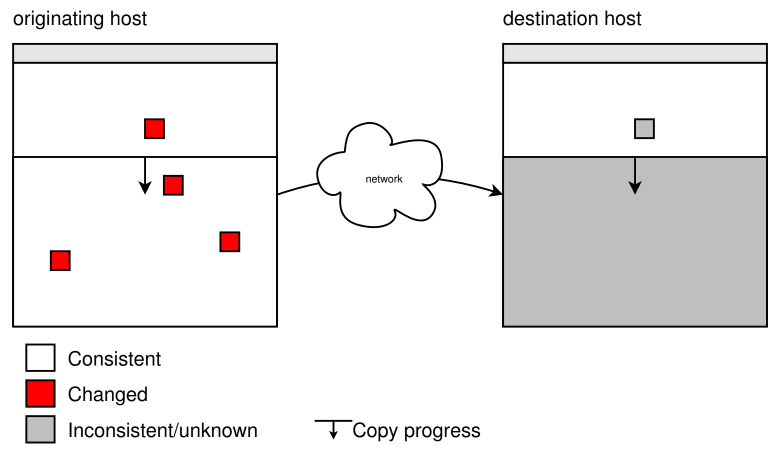
A traditional orthogonal checkpointing algorithm creates a snapshot of a running program. This is relatively simple if the checkpointee (the program being checkpointed) is subject to the control of an external checkpointer, as the checkpointee can be paused during the checkpoint. If the checkpointee is to be allowed to execute in parallel with the checkpointer, a consistent checkpoint can be obtained by revoking all writable mappings and tracking page-faults, backing up original pages in a copy-on-write manner, as described in Skoglund et al. [2001].

Intuitively, self-migration seems like an impossibility, because it involves transferring a checkpoint of a running and self-mutating system. However, with a simple example in mind, it is easy to convince oneself that self-migration is indeed possible: One way of performing self-migration is to reserve half of system memory for a *snapshot buffer*. A checkpoint can then be obtained simply by atomically copying all system state into the snapshot buffer. Self-migration is now simply a case of transferring the contents of this buffer to the destination host. However, this technique is unattractive for two reasons; because the checkpoint buffer wastes half of system memory, and because it will incur a downtime proportional to the size of the self-migrating system.

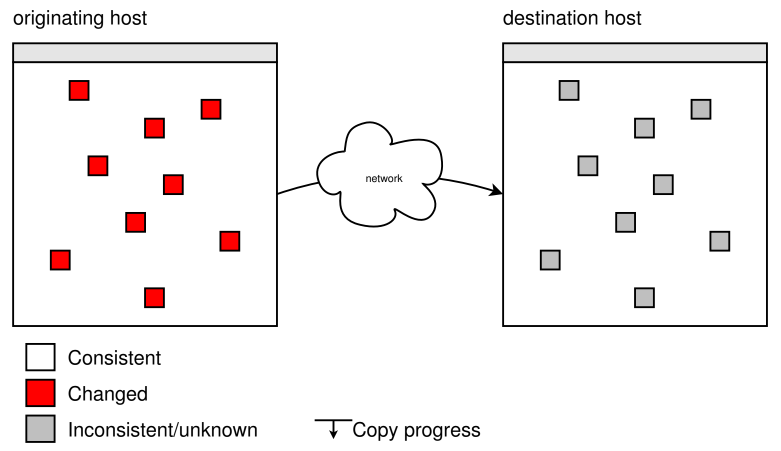
Instead, we use a variation of the pre-copy technique. In the following sections, we de-



(a) Before migration. No consistent state has been transmitted to destination host.

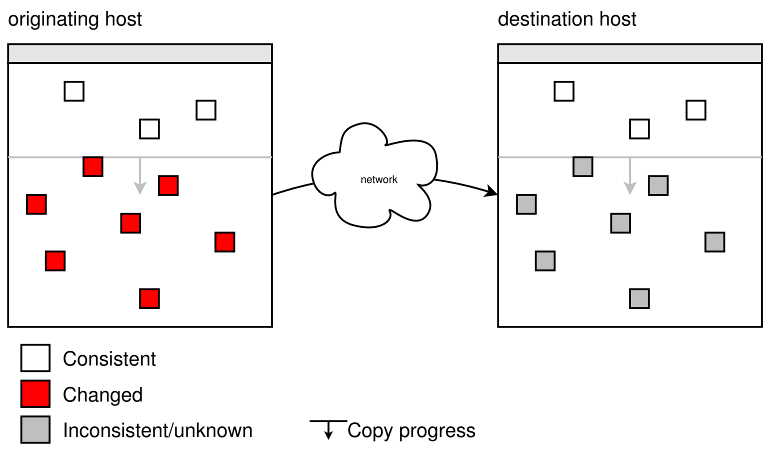


(b) In full-transfer phase. Line with arrow shows progress.

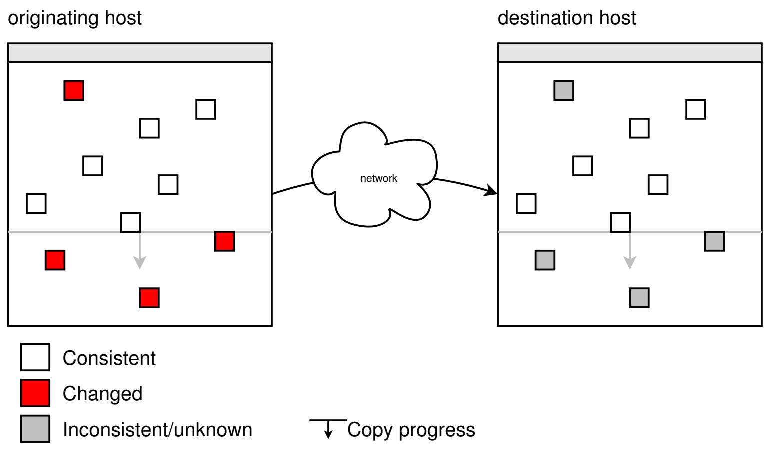


(c) After full transfer phase. Most state has been copied, but mutations on the source have caused inconsistencies on the destination.

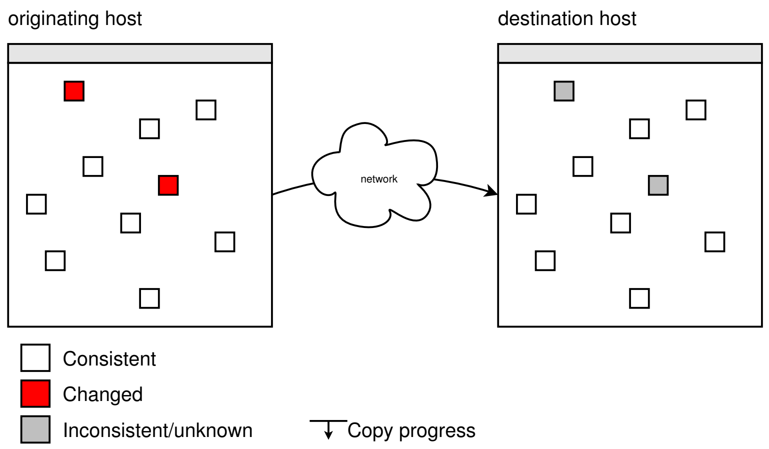
Figure 5.1: Three steps of the Full Transfer phase.



(a) At beginning of first delta phase.

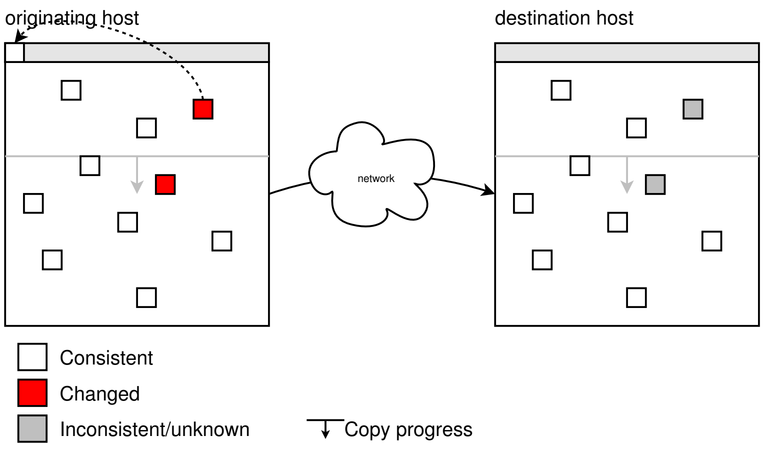


(b) About halfway through first delta.

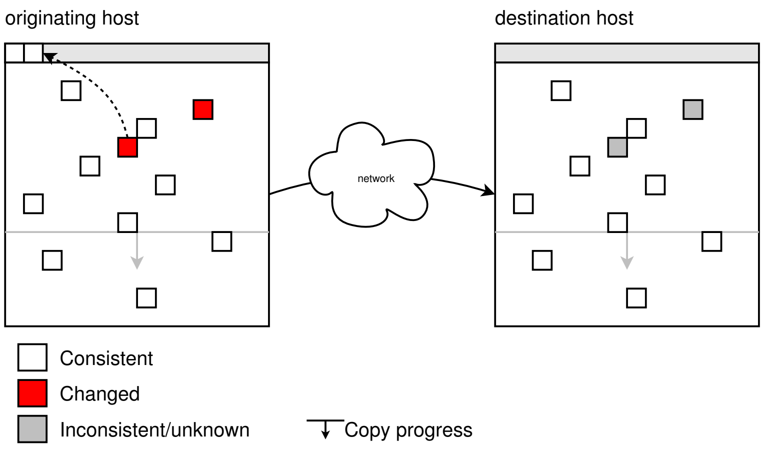


(c) After first delta phase.

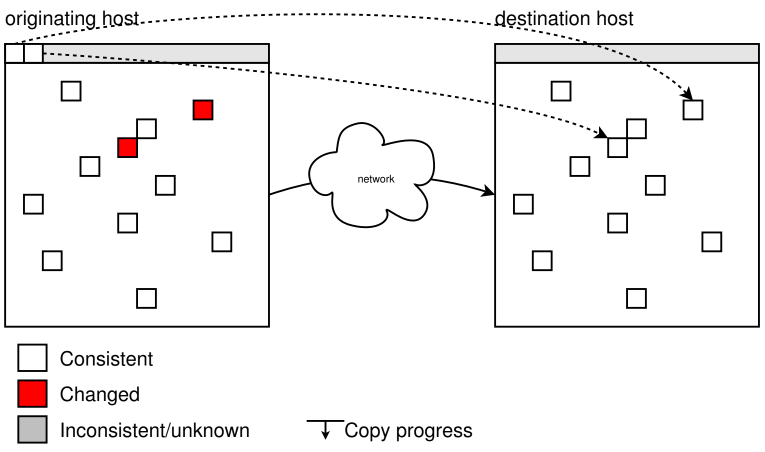
Figure 5.2: First pre-copy Delta phase.



(a) At beginning of CoW phase.



(b) About halfway through CoW phase.



(c) After CoW phase.

Figure 5.3: Pre-copy delta, combined with copy-on-write to snapshot buffer.

scribe in detail the workings of self-migration, as we have implemented it inside Linux running on Xen, and present arguments concerning the consistency of the checkpoint that is migrated.

The writable working set (the pages that are frequently being written) of a running VM is often considerably smaller than the full VM memory footprint. This means that we can start a migration by using pre-copy over a number of rounds, and conclude it by backing up only the remaining set of dirty pages to the snapshot buffer before the final iteration. The size of this buffer can then be reduced to the size of the VM's writable working set, rather than being the size of the full VM image. To save work, we can populate the snapshot-buffer lazily, using *copy-on-write*.

The checkpointing is done iteratively. Initially, all writable virtual memory mappings held by the VM are turned into read-only mappings, and the original value of the writable bit in page table entries is recorded elsewhere, so that page faults resulting from these changes may be discerned from regular write-faults. Figure 5.1 shows the first iteration, where all pages in the VM are copied to the destination host.

Tracking of changes to the checkpoint state is done upon entry to the normal Linux page fault handler. If the page fault is a write fault, and it is determined (by looking at the previously recorded original value of the writable bit) that it is a result of an otherwise legal write, the corresponding page is marked dirty in a bit vector containing entries for all pages in the system.[1] If the checkpointing has entered the final copy-on-write phase, a backup copy of the page contents is made to the snapshot buffer as well.

Our implementation consists of both a user and a kernel space component. Tasks such as TCP connection setup (or, if checkpointing to disk is the goal, opening of an appropriate disk block device), and actual transferral of checkpoint data, happen entirely in user space. The user space process interfaces the kernel space components through a special device node, `/dev/checkpoint`, and reads the checkpoint data from there.

Though the page-fault handler is the main location for tracking of changes to checkpoint state, it is not the only one. All changes to page table contents are monitored, so that during checkpointing only non-writable mappings can be created, and the page tables themselves are marked dirty when modified.

In Xen, guest operating systems see actual page tables containing physical page frame numbers, so page tables need to be remapped to point to different physical pages on the destination node. The checkpoint contains a dictionary mapping old physical frame numbers to offsets in an array of physical pages, and on arrival uses this dictionary for remapping all page tables in the VM.

A final challenge at this point is how to checkpoint state belonging to the guest VM but residing within the VMM or in DMA-accessible memory shared with hardware devices. From a checkpoint-consistency point of view, it is safe to ignore the contents

[1]If the dirty vs. clean state of the page table entry is about to change as a result of the write, the page table page itself is marked dirty here as well.

of any piece of memory if that memory is not referenced from the checkpoint, or if that memory changes without causing any other changes to the state of the checkpoint. When submitting a memory page to the VMM or a device driver for external modification, we remove it from the checkpoint, and when notified by a (virtual) interrupt that the page contents have changed, we add that page to the checkpoint, marking it as dirty.

Self-migration continues over a number of "delta" rounds, where the pages logged as dirty during the previous iteration, are retransmitted to the destination, as shown in figure 5.2. If the page transmission rate is higher than the page dirtying rate of the mutating threads, the deltas will be successively smaller for each round.

In the final round of migration, the last set of dirty pages is copied to the destination. Before a dirty page is sent, the snapshot buffer is checked to see if it contains a pristine version of that page, which in that case is sent instead. Figure 5.3 shows the final phase of migration. Execution can resume later from the checkpointed state, which, we claim, is consistent with the state of the VM before entering the final phase. In the next section, we substantiate this claim in detail.

5.3 Checkpoint Consistency

We can convince ourselves that the above technique results in a consistent checkpoint, by looking at an imaginary Virtual Machine with three pages of memory, a, b, and c. Different versions of page contents are denoted by subscripting, *e.g.*, a_0 and a_1 are subsequent values of page a, and a_0+ is an unknown value resulting from modification of a_0 while it is being copied into the checkpoint. The pages can belong to one or more of the sets S, D, B, and C. S is the set of all three pages of the Source VM, D the set of Dirty pages logged for transmission, C is the current value of the Checkpoint, and B is the set of pages backed up in the snapshot Buffer by copy-on-write.

C is a consistent checkpoint of the system S at time t, if the version number of any page in C equals the version of that page in S at time t.

Initially, all pages are dirty, and the checkpoint is empty:

$$\begin{aligned} t_0: \quad & D = S = \{a_0, b_0, c_0\}, \\ & C = \emptyset, B = \emptyset. \end{aligned}$$

Pages in D are now copied to C, and, as an accidental side-effect of this copy operation, b and c become dirty:

$$\begin{aligned} t_1: \quad & S = \{a_0, b_1, c_1\}, D = \{b_1, c_1\}, \\ & C = \{a_0, b_0+, c_0+\}, B = \emptyset. \end{aligned}$$

We could continue copying dirty pages in D to C, in the hope that the working set would shrink further. However, we are satisfied with a working set of two pages, so

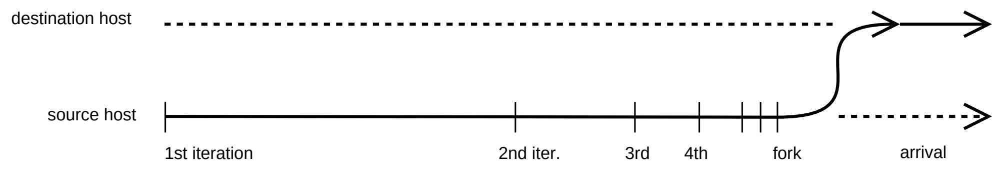


Figure 5.4: Forking execution. After a number of delta iterations, control is switched from the source host below, to the destination host above. There is an overlap of execution in the final phase, that, if not handled, will result in external inconsistencies.

we decide that the state of S at time t_1 is what we want to checkpoint.

We now activate the final phase, copy-on-write, so that dirtied pages get backed up into B before they change. After this iteration, where we again copy pages in D to C, and only c is accidentally dirtied by the copying, we have:

$$\begin{aligned} t_2: \quad & S = \{a_0, b_1, c_2\}, D = \{c_2\}, \\ & C = \{a_0, b_1, c_1+\}, B = \{c_1\}. \end{aligned}$$

By transferring all pages in B to C, we end up with the desired checkpoint:

$$t_3: \quad C = \{a_0, b_1, c_1\}.$$

corresponding to the value of S at time t_1.

5.4 External Consistency

After the activation of copy-on-write at the start of the final phase execution of the migrating VM diverges in two directions, as shown in figure 5.4. The original VM copy keeps running on the source node, while the migrated VM version, based on a slightly out-of-date checkpoint, continues on the destination node. Often, the original copy will be shut down shortly after migration completes, but because it is responsible for completing migration after copy-on-write is activated, some overlap of execution is necessary. If not handled properly, this leads to *external* inconsistencies. This we handle inside the VM in a very simple manner; all outgoing packets with destination IP address and port number different from those involved in the migration are dropped after the activation of copy-on-write, preserving external consistency during and after the migration.

5.5 Self-migration Prerequisites

We have shown how is is possible to self-checkpoint a running program, even one that has the complexity of an entire operating system with multiple, concurrent subprocesses inside. Apart from the somewhat vague statement that "this works in a virtual machine", is worth considering the exact prerequisites of self-migration, *e.g.*, to find out is the technique would be applicable in other process models, such as the one found in UNIX systems.

Based on our implementation of self-migration in the Linux kernel, we identify the following prerequisites:

(a) The ability to trap and handle page faults.

(b) The ability to log all peripheral-induced changes to memory state (DMA).

(c) The ability to write-protect all virtual memory mappings, atomically.

(d) Some way of externalizing checkpoint state, *e.g.*, to a disk or network device.

In UNIX-like systems, it is possible to write protect memory using the `mmap()` system calls, and possible to handle page faults with a signal handler. Simple forms of I/O, *e.g.*, using `open()` and `read()` can also be intercepted and logged. More problematic are features such as shared memory, where an external process may change memory state without notification, and shared files where one-copy semantics mandate that the results of a write are immediately visible to all readers of the file. Finally, the need for atomic write-protection of all virtual memory is problematic in a UNIX-like OS, as this would entail having the ability to stop scheduling of program threads while walking the address space and protecting memory. An alternative here would be system call for revoking all writable memory mappings in a single operation, as is present in the L4 micro-kernel.

Various attempts have been made at freezing process state in UNIX systems, without kernel extensions. The common solution is to somehow force a process core dump, and to later restore that into a running process again. The problem here, as with traditional process migration, is that a UNIX process is likely to make use of, *e.g.*, the file system or shared memory, and so large amount of residual dependencies are likely to exist.

Self-migration can be implemented with relatively simple support from the hosting platform, and in theory should not even require a virtual machine, but could be made to run directly on the native hardware.

5.6 Chapter Summary

In this chapter we have described both hosted migration and self-migration. Because hosted migration has been previously described elsewhere, the focus has been on explaining the self-migration technique in detail. We have argued that the self-migration algorithm is able to obtain a consistent checkpoint of a runinng system, and substantiated this claim by manually performing the self-migration steps on an imaginary system. The actual self-migration implementation is described in more detail in chapter 6.

CHAPTER 6

Implementation

This chapter describes in detail the implementation of our self-migration and checkpointing implementation. We implemented Self-migration in a Linux kernel, running on top of the Xen virtual machine monitor.

6.1 The Xen Virtual Machine Monitor

Xen [Barham et al., 2003] is an open source virtual machine monitor, originally developed at Cambridge University. In contrast to competing commercial offerings, Xen was originally only designed to host para-virtualized guest operating systems, that is, operating systems adapted to run under a VMM instead of directly on the hardware. This design choice allowed certain shortcuts to be taken, with the most prominent one being the omission of shadow page tables. Shadow page tables are an indirection mechanism that allows the guest VM to program its page tables using logical, guest-local, page frame numbers, instead of physical, machine-global frame numbers. The use of shadow page tables simplifies guest implementation, but may be costly in terms of performance, and difficult to implement. Shadow page tables also increases the memory requirements of the VMM, and leads to a trade-off between performance and memory overhead, because a shadow page table that is too small will result in trashing.

In Xen, paging cannot be disabled, and a Xen guest VM needs to have at least one page table describing its virtual address space. If the guest were able to directly write page table contents, it would be possible to create a virtual mapping of memory belonging to another VM or the VMM, bypassing address space protection. The way that Xen solves this problem is by tracking which pages are used for holding page tables, and allowing such pages to be mapped only read-only. The page tables are accessible for reference purposes, but cannot be changed without permission from the VMM. Page tables can only be modified by invoking a VMM hypercall, allowing the VMM to check the safety of the write operation before it is performed. In this way, page tables are maintained as part of the VM, but the VMM controls which pages the VM can map and with what permissions.

Shadow page tables were added to Xen to support live migration. With the advent of Intel and AMD processor support for virtualization, Xen gained the ability to host unmodified guest operating systems, though para-virtualized guests still perform better. As the VM hardware extensions from Intel and AMD mature and translation of shadow page tables is added to the hardware, the need for para-virtualization is likely to be reduced.

Xen hosts a number of virtual machines. The first VM that boots is privileged, and in some ways similar to the UNIX root user, because it holds absolute powers over the physical machine and over other VMs running on it. This VM is known as Domain-0. In the original version of Xen (the 1.x series), the purpose of Domain-0 was purely control, such as the creation and destruction of unprivileged VMs (known as Domain-U's). In Xen 1.x, device drivers were compiled into the VMM, and made accessible to guests through idealized, abstract interfaces. To avoid the burden of maintaining these, the Xen developers decided to instead implement an IPC mechanism for cross-domain communication, and let the privileged Domain-0 provide driver access for the unprivileged VMs through this mechanism [Fraser et al., 2004]. The Xen 2.x and 3.x series thus require the presence of a full Linux VM in Domain-0, to provide the system with device drivers.

6.2 Self-Checkpointing

The overall goal of the self-migration algorithm is to make a consistent copy (a checkpoint) of all in-memory system state, while the system is still running, and this is achieved by using a variation of the pre-copy algorithm, in combination with copy-on-write to a small snapshot buffer in the final phase. While the checkpoint is being obtained, the impact on overall system throughput should be limited, and in case of migration, the "freeze" downtime during which final state changes are copied, should be as short as possible. Throughput impact and final downtime are minimized by running the checkpointer in parallel with the application and system threads of the system being checkpointed. While the system is running, all changes to checkpoint state are tracked using page-fault logging.

One or more *page cleaner* threads attempt to write checkpoint data to external storage, concurrently with a number of *mutating threads* that modify the system state, creating new work for the page cleaners.

Checkpointing happens transparently, in the background. The mutating threads execute legacy software and are unaware of the fact that checkpointing is taking place. Changes to checkpoint state are being tracked by first revoking write-access to all of system memory, and then logging all page write faults as they occur. When a write to a page has been logged, the page is made writable again, to allow the mutating thread to make progress.

When all pages have been copied to the checkpoint, the memory is again write-protected, and the process is repeated over a number of *delta phases*, until the set of pages that change during an iteration (the Writable Working Set (WSS)) stops shrinking. At this point, the pages remaining in the WSS are backed up in a snapshot-buffer, and then copied into the checkpoint.

6.2.1 Write-protecting Memory

Our implementation uses the memory management hardware present in modern CPUs to aid in incrementally obtain a checkpoint of the entire system.

System memory is addressed in fixed-sized chunks, called pages. Page sizes may vary from as little as a few kilobytes up to several megabytes. Smaller page-sizes mean finer addressing granularity, but increase the need for metadata to track ownership and use. In the case of Xen on x86, the only page size supported is 4kB. Figure 6.1 shows a small page table tree. If the CPU tries to reference a virtual address for which no page table entry is present, or if it is present but not writable and the access is a write, the memory management hardware triggers a page fault exception. The exception handler is invoked with information on the faulting address and the type of fault, and can then take appropriate action.

The first step of self-checkpointing is to write-protect all memory, so that changes by the CPU are unable to slip through unnoticed without first causing a page-fault. In an OS such as Linux, every process has a virtual address space, described by a set of page tables. Page tables are implemented differently on different hardware architectures, but a common solution is a radix-tree like data structure that maps virtual addresses into page frame numbers and protection flags. The root of the tree is pointed to by a special *page-table base pointer* register, and for every memory access the memory management unit (MMU) performs a lookup to find out which physical page to read or write from, or whether or not to cause a page fault trap if no mapping exists, or if the mapping protects against a write access.

Because page table address translation involves extra memory accesses, it will be very slow unless page table entries are cached. The Translation Lookaside Buffer (TLB) caches page table entries for a small set of recently used virtual addresses. In a system, such as the Intel x86, with a hardware-loaded TLB, TLB entry loading and replacement is performed transparently, outside of software control. It is thus important to flush the TLB at the relevant times, to ensure consistency with the page tables in memory.

Write-protecting all virtual memory means that the list of all address spaces has to be walked, and all page tables traversed to remove any writable mappings. For the sake of consistency, it is important that causality is preserved, so that all state changes are clearly defined as having happened before or after the memory was write-protected. The write-protection thus has to either happen atomically for all address spaces, or atomically for each address space, right before the address space is next used, *i.e.*, when the page table base pointer register is loaded. The first solution is simpler to implement, but does cause a notable drop in throughput when traversing a large number of address spaces at once. This drop is visible as a drop in bandwidth for an *iperf*

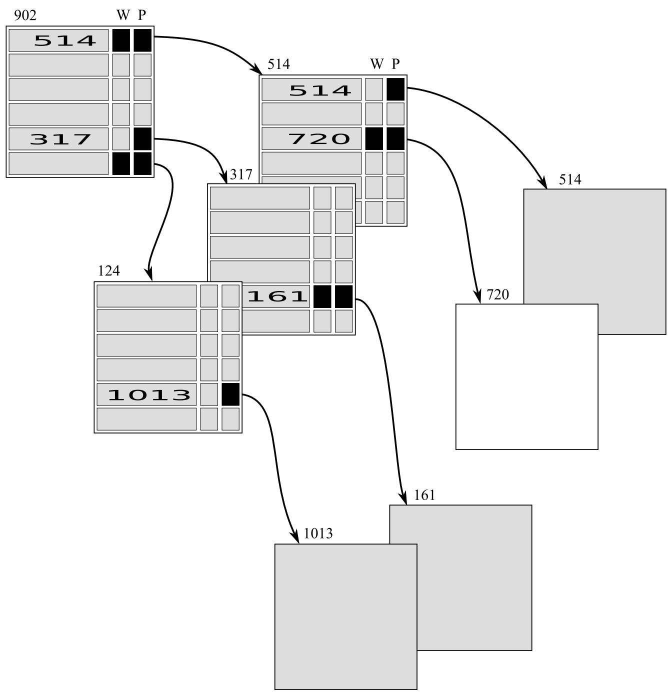


Figure 6.1: A small hierarchical page table tree, with writable (W) and present (P) bits. The page with number 902 contains the page directory. Page 720 is writable, and the rest of the pages are read-only.

TCP-bandwidth measurement, shown in figure 6.2. Another performance optimization would be to only write-protect entries the top-level page directory, postponing processing of leaf page tables. Finally, page tables could also be unmapped on demand, before an address space switch.

The active stack used during unmapped has to remain writable, for two reasons. First, because the unmapping procedure needs a stack, and will immediately cause a write-fault if it stack is unmapped, potentially causing consistency issues, and second, because Xen requires the active kernel stack to always be writable, to prevent an irrecoverable double fault. Thus, the unmapper skips over the current stack pages, as well other special pages only used for checkpointing. The user-space part of the checkpointer can also allocate such special pages, which may be used for holding temporary data during checkpointing. This optional optimization reduces the size of WSS, and thus the final freeze time.

6.3 Page-fault Logging

A central part of the self-checkpointing implementation is tracking of changes to the checkpoint state. The pre-copy algorithm obtains a consistent checkpoint by iteratively logging changes, and updating the checkpoint with new values as changes occur.

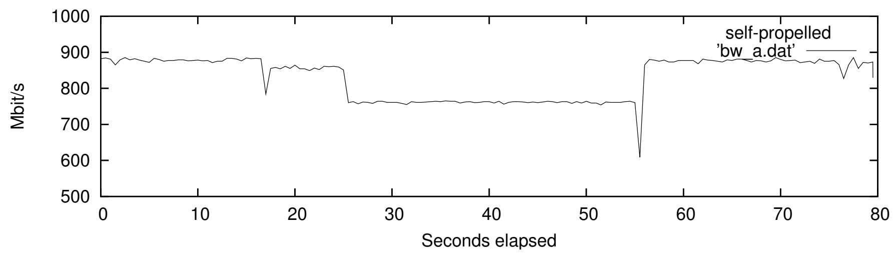


Figure 6.2: Throughput profile of a migrating VM running iperf. The first drop in bandwidth near $t = 17s$ is due to write-protection of all address spaces.

On a machine with paging hardware, this can be done by write-protecting all virtual memory mappings, and then tracking page-faults caused by writes to these virtual addresses. It is also necessary to have a way to discern these write-faults from regularly occurring faults, *e.g.*, faults that are due to normal copy-on-write or illegal accesses. The simplest way of achieving the latter is to reserve one of the unused bits normally present in page tables entries, to store the original value of the page-writable bit. If no such bits are present, this information will have to be stored in an external data structure.

During checkpointing, a new page fault handler that logs checkpoint-faults and forwards regular faults to the normal page-fault handler, must be installed. The page fault trap handler is usually part of a global trap-table, and can be overriden to temporarily point to a checkpointing-specific handler. Figure 6.4 lists an abbreviated version of the page-fault handler installed inside Linux when self-checkpointing is activated. The handler is invoked by the hardware (or the VMM), with an error code describing the type of the fault, and the virtual address at which the fault took place, supplied in registers. The page table mapping the virtual address can be calculated from the fault address (the exact method depends on the operating system), and by looking at the error code at and some of the flags in the page table entry, it is possible to determine if this is a write fault that has occured due to checkpointing. If that is the case, the page frame backing the virtual address is logged as "dirty" for re-transmission, and if not the normal fault handler is invoked instead. While the fault is being handled, the faulting thread is blocked. If the fault handler returns without changing the condition that caused the fault, the thread will resume at the fault location, and thus incur the same fault over and over, without making progress. To allow the thread to continue, the write protection of the virtual address must therefore be lifted, before returning from the fault handler. A final detail is that the page table entry itself may change as a result of the fault-causing write operation. Page table entries contain both a "dirty" and an "accessed" flag, which the paging hardware will set in response to write and read operations. The operating system may use the flags to guide paging decisions, and for correctness they should be tracked as part of the checkpoint. Linux does not use the accessed flag, but the dirty flag must be preserved, so in case this flag is not already set, the mapping page table must also be logged as dirty.

6.3.1 Checkpoint Page Cleaner

The real workhorse of the migration or checkpointing implementation, is the inner loop than takes care of copying pages from main memory and to the (possibly remote) checkpoint storage. This loop is similar to the *page cleaner* often using in database management systems (DBMS). A DBMS page cleaner [Wang and Bunt, 2002] is responsible for paging out dirty memory pages to stable storage, so that the memory pages are free to use for other purposes. There is essentially a race between the cleaner and other, mutating threads, but if the page dirtying rates of the other threads exceed the cleaner's write bandwidth, the cleaner can force the mutating threads to block, making their I/O synchronous with that of the page cleaner. Because of the similarities with a DBMS page cleaner, we adopt the terminology, and refer to the self-checkpointing inner loop as the checkpoint page cleaner.

The page cleaner is controlled by a bit vector with an entry for every physical memory page in the system, where set bits mark dirty pages queued for transmission. Initially, all pages (with the possible exception of free pages, or pages containing state only needed for the checkpoint—*e.g.*, the bit vector itself), have their bits set. The first iteration thus starts out performing a full transfer of the entire system state. Every time a full page has been successfully transferred to the checkpoint, the corresponding bit is cleared from the bit vector, and if the state of that page later changes, *i.e.*, becomes inconsistent, the bit gets set again.

The copying of a page queued for transmission in the previous round may be postponed if the page is dirtied again before it has been transmitted. This reduces the need for repeatedly copying "hot" pages that are constantly changing, and thus reduces the total number of pages transmitted. This optimization does however require the use of two bit vectors instead of one. One vector controls which pages get transmitted, and the other records faults to newly dirtied pages. After each transmission round, the vectors are swapped, so that the dirty vector is now the transmission vector. Pages get added to the dirty vector upon a write fault, and removed from the transmission vector once transmitted, or when the transmission is postponed due to renewed dirtying.

Naturally, sets of pages can be represented by other data structures than bit vectors. In our implementation, bit vectors were chosen for their simplicity and predictable space requirements. For a comparative study of migration algorithms using more complex data structures, the reader is referred to [Noack, 2003].

In addition to the dirty and transmission sets, we keep a set of pages of which copy-on-write copies have been made to the snapshot-buffer, during the final delta iteration. When transferring pages to the checkpoint, this set is first checked to see if a pristine copy of the page should be present in the snapshot buffer.

Memory for the snapshot buffer is currently obtained from the VMM, by the VM growing its allocation slightly. This is to not create additional memory pressure inside the VM, which could alter the cause of execution. If memory from the VMM is not avail-

```
struct page {
 u32_t frame_no;
 char pagedata[PAGE_SIZE]; };
int sock = socket(...);
int f = open("/dev/checkpoint",
                  O_RDONLY);
do {
 struct page p;
 bytes = read(f,&p,sizeof(p));
 if(bytes<0) {
  printf("we␣have␣arrived\n");
  exit(0);
 }
 write(sock,&p,bytes);
} while(bytes>0);
sys_reboot();
```

Figure 6.3: User space migration driver. Because the kernel forks below the user space process, the program will terminate with both the exit() and sys reboot() statements, depending on the side of the connection. Implementing network forks or checkpointing through this interface is straightforward.

able, VM system memory will be allocated instead. If insufficient memory is available, self-checkpointing will abort with an error code.

6.3.2 User-space Control

Initial versions of the self-migration implementation added an extra kernel thread to the guest Linux kernel. This thread would open a TCP connection to the destination host, and transmit the checkpoint data. After resumption on the destination host, this thread would block forever, while the originating guest OS instance would simply halt its virtual machine. Due to this behavior, normal threads and processes were allowed to continue execution on the new host. However, this approach was inflexible, as most of the migration parameters (TCP connection, unencrypted, halt source OS after completion) were hard-coded inside the guest OS. Instead, a new device node `/dev/checkpoint`, from which a user process can read a consistent checkpoint of the machine, was implemented in XenLinux. The data read from this checkpoint corresponds to the data transmitted by the special kernel thread when migrating, a list of (page frame number, page data) pairs, with each page in the system appearing one or more times. The user process performs the actual socket setup, and writes the checkpoint data to this socket. The device `read()` operation signifies end-of-file in different ways, depending on which side of the network fork it is running, so that the appropriate action can be taken. On the source host, `sys_reboot()` is called to destroy the guest instance, and on the destination host a simple arrival message is printed to the user. With this approach, most migration choices are deferred into user space, enabling a user to tailor the migration protocol, and to implement *network forks* simply by not destroying the source instance, and reconfiguring the IP address of the destination instance upon arrival. Figure 6.3 shows the main loop of the user mode migration driver.

6.3.3 Checkpointing the Current Process

The process reading the checkpoint device node needs special care, if it is to survive through checkpointing and recovery. The main reason is that part of this process' state is kept in CPU registers, which need to be recorded in the checkpoint. When the checkpoint is taken, the checkpointer calls a special procedure called `checkpoint_fork()`. This procedure pushes all registers to the process' kernel stack, atomically backs up the kernel stack pages in the snapshot buffer, records the value of the stack pointer register in a global variable, and returns. The checkpoint now contains consistent values of current stack, as well as register information that can be restored to the CPU on resume. After the active process has been forked, checkpointing switches to copy-on-write mode for the remaining pages, rendering the checkpoint immutable. On resume, the stack pointer is restored from the global variable, and the recovery routine jumps to a function that pops the registers off the stack, and returns. The active `read()` system call returns with a special error code, and the process and the rest of the system then resume normal operation.

6.4 Receiving the Checkpoint

Our original implementation did not solve the problem of TCB reduction when receving incoming, self-migrating VMs. Instead, we implemented a TCP-server running as a privileged user process inside the Domain-0 VM. Upon receiving a connection request, the privileged migration server would create a new VM. Incoming pages would them be copied to the VM, and after receiving the last page and meta-data such as the values of the stack pointer and program counter, the VM would be started. Due to the problems with page tables that we will describe in section 6.6, each incoming page would be annotated with a type identifier, and page-tables were rewritten on their way in.

The problems with this approach were twofold; it depended on the existence of a full TCP/IP stack in the privileged control plane software, and it hard-coded the migration protocol, hindering future evolution. A full TCP/IP stack in the control plane would defeat one of the purposes of the self-migration algorithm, namely the ability

```
void page_fault_checkpoint_preample(
  unsigned int virtual_address,
  unsigned int error_code) {

  /* does error code mean 'write to write-protected,
     present page'? */
  if(write_to_present_page(error_code)) {
    pte_t* pte;
    pte_t pt_bits;
    unsigned int chkpoint_present =
          _PAGE_CHKPOINT|_PAGE_PRESENT;

    /* get pointer to page table entry for fault address */
    pte = lookup_pte_for_virtual_address(virtual_address);
    pt_bits = *pte;

    /* is this a write fault caused by checkpointing? */
    if(( pt_bits & chkpoint_present)==chkpoint_present ) {
      pfn_t page_frame_number;

      page_frame_number = extract_pfn_from_pt_bits(pt_bits);

      /* log the page frame to which the fault referred as
         dirty (in need of retransmission) */
      log_dirty_pfn(page_frame_number);

      /* if this write changes the contents of the page table,
         we also have to log the page tables as dirty */
      if(pt_bits & _PAGE_DIRTY == 0)
        log_dirty_pfn( virtual_to_pfn(pte) );

      /* make mapping writable, to allow the faulting
         process to continue */
      *pte = _PAGE_RW | (pt_bits & (~_PAGE_CHKPOINT));

      /* pretend this page fault never happened */
      return;
    }
  }
  /* this was a normal fault, forward it to Linux'
     own page fault handler */
  do_page_fault(virtual_address,error_code);
}
```

Figure 6.4: Abbreviated page-fault handler preamble. This has been adapted from the actual implementation for ease of understanding. Type and function names have been adjusted, and code for locking etc. has been left out.

to migrate without outside assistance, and could prove to be an Achilles' heel to security, due the added complexity. In addition, the protocol would have to anticipate any future uses, including support for different encryption and compression types. In addition, the initial implementation was not designed to scale to multiple clients simultaneous migrating into the machine, and in a production system this is something that would have to be added as well, increasing complexity.

The privileged server only allowed resuming incoming checkpoints for migration, but not bootstrap of packaged guest OS images, another feature that would have to be added for production. While this is a feature one could add, the problem is that different operating systems come in different binary formats, and all of these would have to be supported.

Instead, we moved to a model that is similar to the bootstrap process of real machines. The typical way a computer bootstraps is that the CPU jumps to an address in the machine firmware (ROM or Flash memory), and that code there understands enough about the hardware to load a simple boot-loader from the disk. The boot-loader is installed by the user or along with the operating system, and understands the binary format of the OS kernel binary, which it loads into memory, decodes, and calls. At this point the OS takes over and handles the remainder of the bring-up process.

In our model, we instantiate a small VM with pre-installed "firmware", containing enough functionality to fetch or receive a boot-loader binary from the network. It is this boot-loader that parses the input data format, and can boot or resume it into a full running VM guest. By adopting this two-stage design our system became more flexible, because the boot-loader is supplied by the client rather than by the control plane, and more secure by moving functionality from the control plane to the VM.

6.5 Resuming the Checkpoint

Recall from section 6.1 that Xen requires paging to be enabled at all times. In order to receive checkpoint data from the disk or network, virtual mappings for checkpoint pages therefore have to exist. In this case, a few temporary mappings will suffice, but before control can be turned over the resume procedures inside the checkpoint, page tables must be set up to correspond to at least a part of the virtual address space layout expected by the checkpoint. Once control has been turned over to a procedure inside the checkpoint, this procedure can exploits its access to internal data structures in the checkpoint to perform the final steps of recovery, back into a fully running guest operating system.

6.5.1 Unroll Phase

After all checkpoint data has been received inside an otherwise empty VM, the checkpoint must be resumed to a state where it can be called, *i.e.*, has a valid virtual memory layout to the degree that program and stack pointer can be safely pointed to addresses inside the checkpoint. We refer to this as the *unrolling* phase, different from the *recovery* phase occuring next. The purpose of unroll is to initialize CPU and MMU hardware to a level from where the checkpoint may handle the remainder of recovery. As described, we are using a two-stage boot-loader to both boot fresh guest OS images, and for resuming checkpoints. In the second stage checkpoint loader, we unroll the checkpoint by creating temporary page tables from which a part of the checkpoint can run, and perform the final recovery. The unroll phase ends by the loader turning over control by jumping to a procedure inside the checkpoint. The last step is not entirely trivial, because it requires that there is a virtual memory overlap between the loader and the checkpoint address spaces. Because the checkpoint loader is Linux-specific, we can solve this by exploiting knowledge of the layout of the Linux virtual address space.

6.5.2 Recovery Phase

Code executing at the recovery phase resides inside the checkpoint image, and has access to the kernel part of the checkpoint virtual address space. This code executes with capabilities similar to those of an interrupt handler—it has access to all kernel data structures, but cannot block or perform any other activities related to scheduling, and it cannot access user space parts of the virtual address space. Its main responsibility is to remap all process address spaces, as described in section 6.6, reinitialize all global CPU state (segment descriptors, debug registers, page table base pointer), free all memory used for self-checkpointing (such as snapshot pages), and finally to turn interrupts back on and resume the guest OS to full multi-user mode.

6.6 Remapping Identifier Spaces

In a "pure" or fully virtualized virtual machine, all identifier names are properly nested within the VM. This means that the VM is able to refer to all resources using local names, and that in the case when two VMs use the same name for a resource, those name refer to distinct resources, rather than the same one. Such namespace-nesting prevents future problems with name collisions, *i.e.*, if a VM is moved to another set of page frames, physical CPUs, or disk blocks, or if a VM is migrated to a different physical machine, where the exact same set of resources is unlikely to be available. Performance, however, is often in conflict with the goal of namespace nesting. Xen, for instance, is not a pure VM, and allows VMs to refer to page frames using real hardware page frame numbers. This design choice does away with the need for a layer of indirection between VM and hardware, and has been critical in achieving good run-time performance for workloads that excercise the paging hardware. Though not the only example of unnested namespaces in Xen, this is the main point at which Xen differs from earlier microkernels such as L4, as well as from the competing VMWare product.

When migrating VMs between machines, this unnested resource naming problems must be addressed, as the unrolled VM will be unable to continue on the destination node using resource names pertaining to the source node. In essence, either the VMM or its page allocation policy must be modified to provide proper nesting, or to guarantee that an incoming VM will receive the exact same set of page frames it possessed on the source node, or all page tables in the VM have to be translated from the old to the new set of page frame numbers.

6.6.1 Page Table Indirection

The L4 microkernel abstracts page tables through a kernel API which lets unpriviliged process threads update page tables inside the L4 microkernel. If the unpriviliged process hosts a guest operating system "personality", such as L4Linux, which expects to be able to manipulate page tables contents directly, this approach leads to duplication of page table contents between guest and microkernel, and thus waste of memory as well as additional communication between microkernel and guest OS, to keep the two copies synchronized. Worse, this opens the door to a Denial-of-Service (DoS) attack in which the guest is able to consume all microkernel memory by building a very large virtual address space, backed by only a few real memory pages. L4 attempts to counter this type of attack by restricting the number of times a real page can be referenced in the same page table, but this policy leads to other problems that an application programmer has to work around. Recent revisions to the L4 specification require applications to provide pages for holding page tables from its own pool of pages, in which shadow copies reside.

A more scalable approach is to treat in-kernel shadow pages as a cache of translated page tables. This approach is taken by VMWare, as well as Xen when in "shadow mode", used when hosting unmodified guest OSes and when transparently live migrating VMs. Because the size of the cache is no longer controlled by the guest, a single client is no longer able to DoS the system by consuming all page table resources, but the cache is likely to become a contention point for workloads with large working sets.

In both cases, the additional book-keeping required for keeping shadow page tables up to date with the guest VM page tables, will incur some runtime overhead. Xen pioneered the use of "direct" page tables, where the guest keeps control of the page tables, but has to call on the VMM for every update. This allows the VMM to maintain the invariant that only pages belonging to the VM are referenced in its page tables.

6.6.2 Cross-node Identical Reservations

In the case where each node hosts only a single (guest) OS instance, it is reasonable to assume that the identical set of machine page frames will be available at the destination node. Frequently the goal of a VM system is to host multiple OS instances, and requiring cross-node identical reservations in this case will lead to fragmentation and poor utilization.

6.6.3 Rewriting Page Tables

The approach we are adopting is to rewrite page tables during checkpoint recovery. For this to be possible either the page tables must be "canonicalized" during checkpoint creation, or enough information has to be available to convert from one set of machine frame numbers to another. Canonicalization means that all page table machine frames are converted from the machine addresses into logical addresses in the range $[0 : n)$, with n representing the total number of pages owned by the VM. Every time a page table is about to be copied to the checkpoint, a canonicalized version is created by converting each entry through a lookup table, mapping machine to logical page frames. This technique has the benefit of working on the fly, thus reducing overall migration latency, but requires knowledge of which pages are in use as page tables, both during the incremental checkpointing, and at checkpoint recovery.

One of our first implementations annotated the checkpoint with a list of all page tables in the checkpoint, and this information was used when recovering the checkpoint. Two problems with this solution became apparent during development:

Variable checkpoint size When checkpointing to disk, it is desirable to have a one-to-one correspondence between disk blocks and memory pages, to allow for efficient loading of the checkpoint (a straight copy from disk to main memory), and to locate the ith checkpoint on a disk using simple multiplication. If the checkpoint is prefixed with a list of page tables, the checkpoint is no longer of constant size. An array or bitmap describing the type of *every* page in the checkpoint would be constant size, but cost more in terms of storage.

Precise tracking of page tables frames The main problem is that with versions of Xen later than 2.0, it became very hard to reliably track which pages in a guest OS were in use as page tables. Though the guest would notify Xen when an upper level page directory went into use as a page table, no such notification was available for lower level page tables, unless one wished to recursively scan the page table tree for all its leaves.

After experimenting with on-the-fly canonicalization, we ended up simply remembering, as part of the checkpoint, the complete list of machine page frames. Upon recovery, we can convert this list into a dictionary, which we consult when rewriting the page tables for all address spaces in the VM. The list of address spaces is already known (it can be derived from the process table), so no extra information is needed or has to be tagged onto the checkpoint. There is still a boot-strapping problem however, as the recovering VM needs to have a virtual address space set up before data structures such as the process have any meaning. We solve this in the unroll phase by exploiting knowledge of the Linux virtual address space layout. Linux always maps all machine pages contiguously at the top of virtual memory, and by doing the same our checkpoint loader is able to recreate the kernel parts of the Linux address space, before jumping to the recovery procedure inside the checkpoint. We create this mapping using temporary page tables, and the only information needed for this is a list of machine frames belonging to the checkpoint VM (inside which the unroll phase code is also running). The checkpoint loader is Linux-specific and provided as part of the checkpoint, not the pre-installed software base, and separate loaders will have to be implemented for other self-checkpointing operating systems. This approach allows us to target the checkpoint-unroll code to the needs of the speficic OS inside the checkpoint, while staying compatible with loader formats or checkpoints from other OSes.

6.6.4 Verification of Checkpoint Integrity

During development, it is often useful to empirically test the correctness of the implementation, *i.e.*, if the created checkpoint is consistent. A slightly inconsistent checkpoint may be able to resume and appear to be correct, but will at some point crash or start corrupting data. Fortunately, we have a very clear notion of what the correct checkpoint looks like, because we know the state of the system at copy-on-write activation point. If the machine on which we are testing contains enough memory, we are able to make a snapshot copy of the entire system state right before activating copy-on-write. After checkpointing has completed, we can compare this copy with the checkpoint actually written, and report back any discrepancies we might discover. If enough memory is not available, we can store a hash of the snapshot instead, though in this case we cannot pinpoint discrepancies as precisely. This technique does not guarantee correctness, but is a valuable tool during development.

In addition, our user-space migration driver can optionally checksum outgoing checkpoint pages, so that the integrity of a checkpoint sent over an insecure channel can be verified.

6.6.5 Determining the Size of the Snapshot Buffer

The self-migration algorithm works because the writable working set (WSS) is usually only a small subset of the pages in the checkpoint, as this is what allows us to use a small shapshot buffer to preserve consistency in the final delta round. The size of the snapshot buffer is therefore important. If the snapshot buffer is smaller than the WSS, the algorithm is unable to terminate with a consistent checkpoint, and if the buffer is

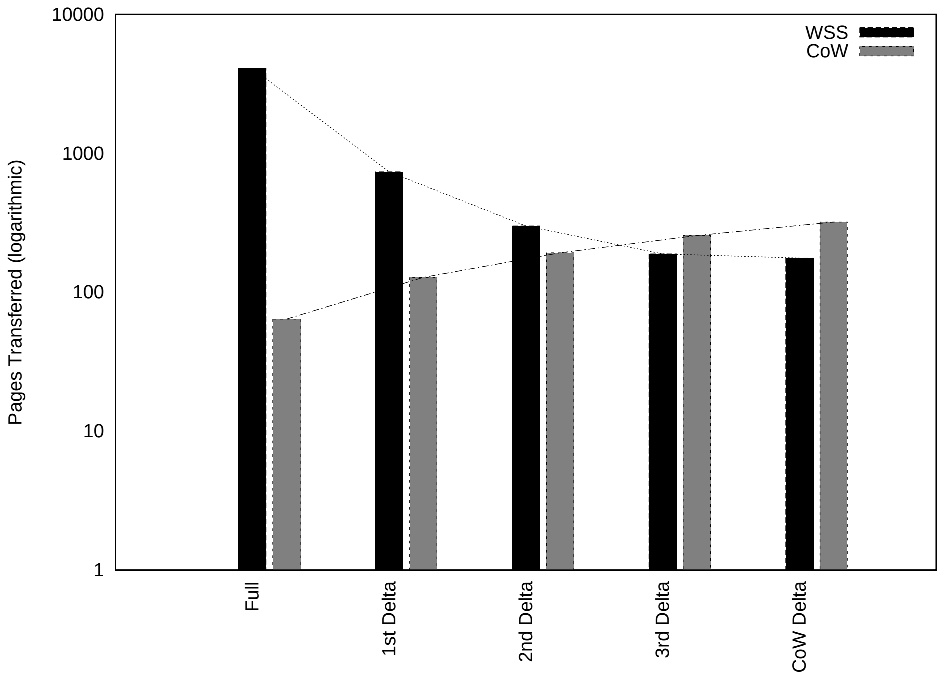


Figure 6.5: Size of Writable Working Set, vs. size of Snapshot CoW Buffer (logarithmic scale).

too large, valuable memory is wasted. In our implementation we adjust the size of the snapshot buffer dynamically, and at the same time we try to adjust the WSS by adjusting how processes get scheduled. The snapshot buffers starts out small (64 pages), and grows linearly for every delta iteration. When the two lines cross, *i.e.*, the WSS becomes small enough to fit the snapshot buffer, the checkpoint is taken. Figure 6.5 shows the development in WSS and snapshot buffer sizes during checkpointing of a small 16MB VM. If the snapshot buffer grows beyond a certain limit, the checkpointer fails.

6.6.6 Adjusting the Writable Working Set

Unlike traditional pre-copy migration, the self-migration algorithm may fail if the WSS is too large, and the page cleaner unable to keep up with the page dirtying rate of the mutating threads. Common workloads are optimized for locality and have modestly sized working sets, but given that the memory bandwidth normally exceeds network or disk bandwidth, writing a program that dirties memory faster than it can written to a checkpoint is trivial.

Situations may thus arise where we ultimately need to force the mutating threads to synchronize with the page cleaner in order to ensure progress, and not have to abort the checkpoint. It is not always possible to directly synchronize, or block, a mutating

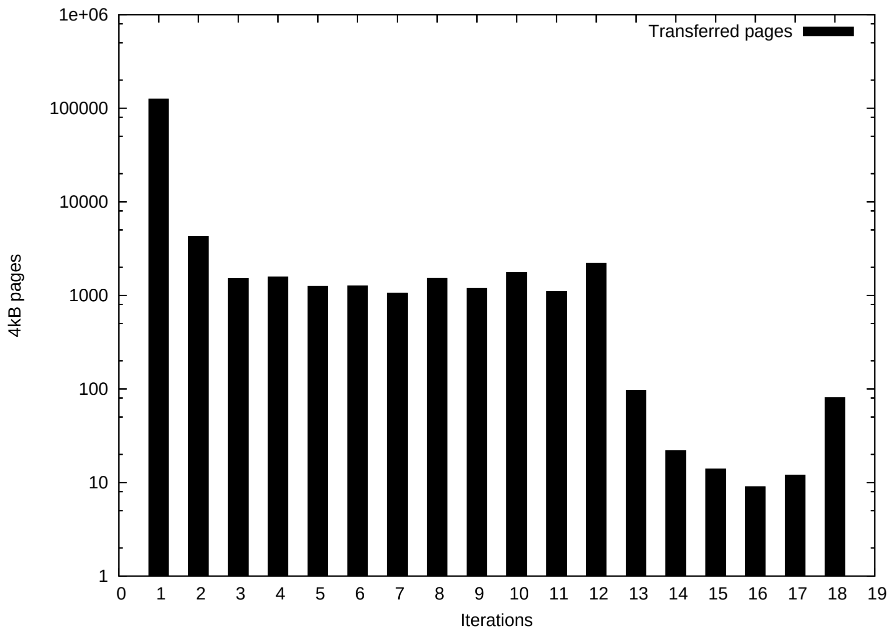


Figure 6.6: Effect of page-fault rate-limiting (logarithmic scale).

thread so that it waits for the page cleaner, but scheduling of processes can be made WSS aware, to reduce the overall page dirtying rate.

The page dirtying rate can be adjusted by taking the WSS of processes into account in the scheduler, or by suspending processes that exhibit uncooperative behavior. We have attempted various implementations of WSS-aware scheduling, using the following approaches:

- Tracking of per-process working sets, and prioritizing processes with small sets over processes with large ones.
- Enforcing an overall credit limit on all processes, and pausing the first process to reach that limit, in the page fault handler. When the page cleaner has made enough progress, blocked processes are awoken, and the system continues as before. In practice, this can be implemented by blocking heavily faulting messages in the page fault handler, adding them to a wait queue from where they can later be woken up. Care must be taken not to block processes while they are inside the kernel, as this will likely result in a deadlock.
- The use of a high-priority idle thread, which tracks checkpoint progress, and yields to the hypervisor instead of to mutating processes, when progress is unsatisfactory.

The first approach will keep processes with small WSSs responsive, but requires changes to the scheduler. The second approach will likely single out large processes, though a small process may be unlucky and get caught instead. The global credit is calculated as the difference between transmitted and dirtied pages, and processes are checked against the credit limit inside the page-fault handler preamble.

The effect of page-fault rate limiting can be seen in figure 6.6. Here, self-migration of a 512MB VM running a Linux kernel compile task is attempted. The task has a large WSS of more than 1000 4kB pages, and insufficient progress is made. After the 12th iteration, a limit is imposed on the page-faulting rate, leading to a drastic reduction of the WSS, and successful migration.

6.7 Overhead of Migration

We measured the overhead added by the self-migration implementation for different types of workloads, one purely CPU bound (POV-Ray), one that is primarily network bound (httperf), and one that both I/O and CPU processing (Linux compile).

For all the tests, the setup consisted of three machines; a 2.1GHz Intel Core Duo 2 CPU, running the actual workload in a Linux 2.6.16 Xen VM, an Intel Pentium 4 at 2GHz, running Xen, and Linux 2.6.16 in the privileged Domain-0, and an IBM Thinkpad T42p sporting a Intel Pentium M processor at 1.80GHz, running a standard Linux 2.6.18 kernel. All machines were equipped with Intel PRO/1000 Gigabit Ethernet adapters, through a Linksys SD2005 Gigabit switch. In the tests, the overhead of checkpointing and migration was measured by repeatedly sending a complete memory-checkpoint to the migration target. The checkpoint was requested from the purpose-written in-VM web server (described in chapter 8), by the standard `wget` command-line HTTP client. Control was never actually turned over to the migration target, so the tests did not measure the cost of re-associating with a new network interface, or the cost of checkpoint recovery. The purpose of these tests was not to measure the performance of the underlying Xen system, as this has been done elsewhere [Barham et al., 2003; Matthews, 2004].

In the first two tests, the Core Duo machine ran the actual workload, the Thinkpad acted as a migration target, and, in the iperf and httperf tests, the Pentium 4 was used for running the benchmark client (`iperf -c` and `httperf` respectively).

As example of a purely CPU-bound workload, we chose the Persistence of Vision Ray Tracer (POV-Ray), which includes a standard scene recommended for benchmarking. The program first reads the input scene description, then pre-calculates a number of photon maps, and finally renders the image in the desired resolution (384x384 pixels). We ran this test in two scenarios, in the XenLinux VM, and in the XenLinux VM while constantly transmitting checkpoints to another machine, over the Gigabit network. From the results in table 6.1 we see that constant migration results in an 18% stretch-out of execution time.

For a more complex workload, a full build of the Linux 2.6.18 kernel source tree, executing under the same conditions, the stretch-out is longer at 31% when measuring wall-clock time (table 6.2.) However, when looking at the fraction of time actually

Scenario	Wall Time		User Time	
Unmodifed Xen VM	49m 44s	100.0%	49m 44s	100.0%
Xen VM with self-migration patch	49m 44s	100.0%	49m 44s	100.0%
Self-migrating Xen VM	58m 35s	117.8%	50m 13s	100.1%

Table 6.1: Completion times for the POV-Ray benchmark.

Scenario	Wall Time		User Time	
Unmodifed Xen VM	3m35s	100.0%	3m10s	100.0%
Xen VM with self-migration patch	3m35s	100.0%	3m10s	100.0%
Self-migrating Xen VM	4m42s	131.2%	3m23s	106.8%

Table 6.2: Completion times for the Linux compile benchmark.

spent in the user programs, we see only a small increase. The reason for this discrepancy is that the Linux compile has a fairly large writable working set, and several of its jobs are subject to page-fault rate-limiting during the benchmark. The additional delay stems from having to wait for the checkpoint page-cleaner thread, rather than from extra page fault processing or other checkpointing overheads. In other words, having a faster network available would likely reduce the stretch-outs caused by migration. These overheads represent the case where the VM is constantly migrating, something that is not likely in a production system. If migration is less frequent, the overhead is going to be less profound, as confirmed by the results presented in chapter 8.

As a baseline, we also ran both benchmarks in a VM without the self-migration patch, and found that, when not active, the presence of self-migration functionality in the guest VM added no overhead for these workloads.

Secondly, we attempted to measure the impact that migration has on network I/O. We measured this using the `iperf` and `httperf` benchmarks. While the benchmark is running, we repeatedly migrate the state of the VM, with a 10 second pause between each migration.

Migration affects the performance of the VM guest OS in two ways; by consuming CPU cycles needed by the benchmark application and guest I/O subsystem, and by consuming actual network bandwidth. To be able to distinguish between these effects, we ran each of the tests in three scenarios; migration using all available bandwidth, migration using adaptive traffic shaping to reduce consumed bandwidth by briefly blocking in the user-space checkpointer between transmission bursts, and finally "null" migration, where the migration user space driver discards checkpoint data and blocks briefly, instead of transmitting. Results that show how migration affects bandwidth over time are shown in figure 6.7 and figure 6.8.

In both tests, we see a clear impact from the actual transmission of checkpoint data, consuming more than half the available bandwidth in the iperf case. A 50% bandwidth drop is not surprising, given that the checkpointing process and the iperf server

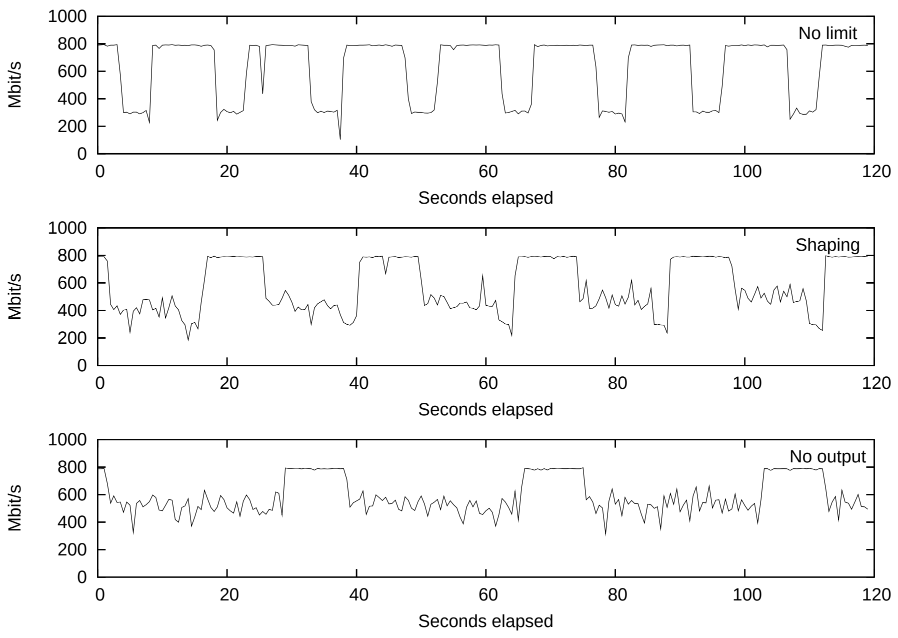


Figure 6.7: Impact of migration on a network-bound iperf process, over a 120 second period.

process compete for the same network resources. For `iperf`, we attribute the remaining difference to the fact that checkpointing process does more work inside the kernel, and thus gains a scheduling advantage, and that the TCP bandwidth measurement is sensitive to timing variances. We also see that it is possible to reduce the impact migration has on the application, by adapting the transmission rate of the checkpointer, though at the cost of a longer overall migration time.

6.8 Current Limitations

The implementation currently does not support VMs with more than a single CPU assigned. The main reason for this limitation is the way that unmapping of addresses spaces works by traversing all address spaces and marking page table entries read-only. Doing the same atomically in a multi-CPU configuration is less trivial, because it will require locking of all address spaces, which may result in a deadlock. A more scalable and correct alternative would be to unmap each address space on demand, before a context switch, either by walking the entire address space, or by installing a shadow read-only page directory, and then unmap leave page tables on demand. We have been experimenting with the first approach, but do not have a stable implementation at this point.

Another potential issue that we do not address, is subtle differences between different processor versions. It is common for the OS to negotiate access to various optional

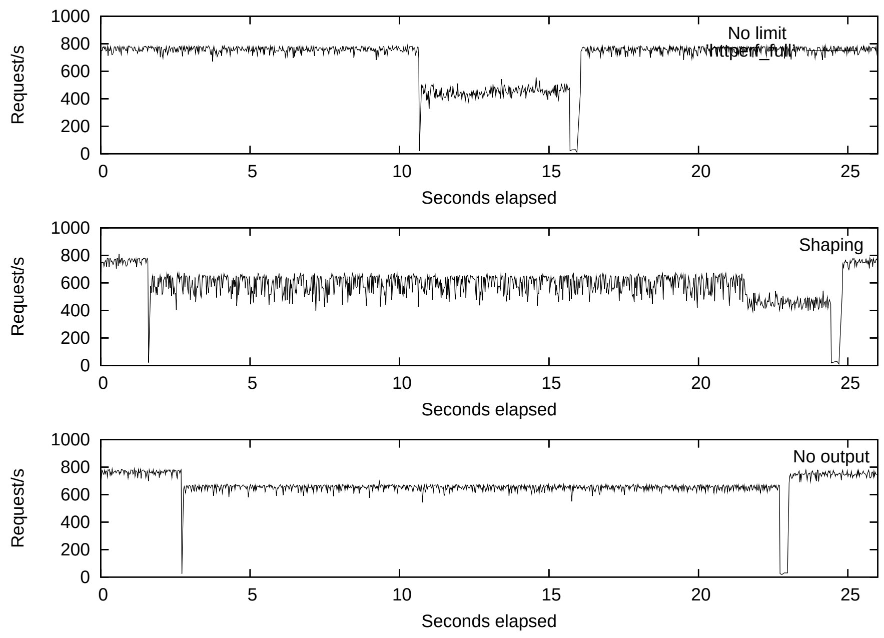


Figure 6.8: Impact of migration on a network-bound web server process, over a 26 second period.

processor features at startup, and when migrating to a new machine it is important to ensure that these features will be present at the destination, or alternatively, to renegotiate them upon arrival. As long as the optional features only are used inside the OS, and not by user-space applications, renegotiation should suffice, but if information about the features has already been passed on to applications, correct execution cannot be guaranteed, and migration to an incompatible hosts should be prevented.

6.9 Chapter Summary

In this chapter we have described the low-level details of our self-migration and self-checkpointing implementation, in Linux 2.6.16 running on Xen 3.0.1.

The four main tasks involved are: Memory write protection, page-fault logging, checkpoint recovery, and page-fault rate limiting. Rate-limiting is necessary because self-checkpointing relies on a limited-size snapshot buffer. We have described techniques for remapping of global identifiers, and techniques for resuming a checkpoint back into running state, and finally we have measured the performance of the implementation for different workloads.

The following two chapters describe methods for dealing with the relocation of a VM to a new network attachment point, and for dealing with data residing on local or network attached disks.

CHAPTER 7

Networking and Storage for VMs

Most VM applications need access to one or more networks, whether for inter-job coordination, client/server communication, or remote management, and access to storage for retrieving input data sets or storing results data. This chapter provides an overview of a number of common approaches to networking and storage for VMs in on-demand computing systems.

7.1 Network Filtering and Mobility

At the physical level, one or a few network interfaces will have to be multiplexed across all VMs on a host, and in addition features such as live migration of VMs between hosts may be desired and should be supported. Below, we will outline four strategies for providing connectivity in a virtual machine-based on-demand computing system, and the various trade-offs involved. The strategies are selected subject to the following constraints:

- All traffic to and from a VM should be filtered, to prevent eavesdropping and forgery. Ideally, it should also be possible to identify the origin of any network packet, to deter antisocial behavior.
- VMs should be able to migrate between hosts, without loss of connectivity or connection state.
- The solution should be backwards-compatible with existing applications.

The strategies that we outline are the following: Ethernet MAC-level filtering, filtering on IP addresses, TCP connection migration, and a combination of tunneling and NAT.

7.1.1 MAC-level Filtering

Most commodity-PC clusters are connected with Ethernet interfaces and switches. Ethernet packets, called *frames*, flow through the local area network guided by 48-bit Media Access Control (MAC) source and destination addresses. Each interface contains a factory-supplied, globally unique MAC address, and incoming frames are filtered by comparing frame destination addresses against the MAC address of the interface. Ethernet was originally a broadcast medium, but due to the poor scaling properties of broadcast, modern versions employ switching instead. Each interface is connected to a port in the switch, and the switch learns the MAC address or addresses used on that port. The reason there may be more than one address in use on each port is that the port may be multiplexed further, *e.g.*, by the use of a hub which is a simple frame broadcasting device.

Modern Ethernet interfaces can be programmed to overwrite the factory-supplied MAC address, and even to filter on multiple MAC addresses in hardware, or alternatively to receive all incoming frames and let the OS do the filtering in software. The Ethernet standard allows the use of custom MAC addresses, through setting of the "locally administered network" bit in the first byte of the MAC address. This functionality can be used for multiplexing traffic, by assigning a synthetic MAC address to each VM using the interface.

The risk with this *data link layer addressing* strategy is that the synthetic MAC addresses may no longer be globally unique, and so the LAN may become destabilised by address collisions. On the other hand VM mobility comes easy, because the switch will quickly learn that a VM has moved to another port, and start switching traffic accordingly. It is also possible to track the origin of any packet, by mapping MAC addresses to VM identifiers.

7.1.2 IP-address Filtering

At the *network layer*, it is possible to assign a unique IP address to each VM, and keep it when the VM moves to another host. Before going into details with this strategy, we are going to briefly describe the structure of IP addresses, and the way that traffic gets routed through the Internet:

Structure of IP addresses

The Internet Protocol version 4 (IPv4) uses hierarchical 32-bit addresses to control the routing of traffic between networks. Originally, the space was split at octet boundaries, and each participant network would be assigned one or more A-, B-, and C-classes, subspaces of 24, 16, and 8 bits respectively, to administer. The IP address space of 2^{32} numbers was considered vast, and the fact that the smallest amount of numbers that could be assigned to one organization was a C-class of 2^8 numbers, did not seem like a problem at the time. IP addresses could be arranged in a tree-like data structure, to provide for efficient routing of traffic. As the Internet grew, addresses became more scarce, and the need for a finer-grained division of the address space led to the deployment of Classless Inter-domain Routing (CIDR) [Fuller and Li, 2006]. CIDR removed the 8-bit space-division boundary requirement, so that some subnets today have as little as 2 or 3 bits of address space. While this has resulted in more complex router implementations, employing route-peering protocols such as the Border Gateway Protocol (BGP) [Lougheed and Rekhter, 1991], the hierarchical name-space prevails and is still a necessity for efficient routing.

Motivated by the scarcity of 32-bit IP addresses, the IETF set out to design a successor to the successful IPv4 protocol. IPv6 extends the IP header address fields to 128 bits, effectively solving the address scarcity problem for as long as the human race is constrained to a single solar system. IPv6 also contains other additions, such as support for MobileIP [Perkins and Myles, 1997], that would make VM relocation more seamless. Unfortunately, IPv6 adoption has been slow, with the exception of countries like Japan where the IPv4 address space scarcity is most strongly felt.

Today it is possible to build a router with a main-memory route entry for each of the 2^{32} addresses of IPv4, and if widely enough deployed, such routers would allow IP addresses to roam freely across the Internet. But given this possibility, a new problem would arise; the ability of attackers to hijack IP address, unless some kind of authentication scheme could be deployed. For this an other reasons, IP addresses are likely to remain hierarchical.

Address Resolution Protocol

IP addresses are not only used when routing traffic between networks, but also for identifying hosts on the local network. In an Ethernet for instance, a host is configured with one or more IP addresses, and even though the Ethernet has no understanding of IP headers, is immediately reachable by IP. The mechanism that makes this work is known as the IP-over-Ethernet Address Resolution Protocol (ARP). The first time a host needs to talk to an IP address on the local subnet, it broadcasts a ARP request, asking the remote party to identify itself. The host with the corresponding IP responds with its Ethernet MAC address, and communication can commence using IP.

VM mobility and ARP

In cases where it is practical for each VM to have its own IP address, it is possible to migrate the VM within the local subnet, without losing network connections. When a VM has migrated, it will stop responding to traffic coming to the original network interface, and this will be noticed by local peers and routers. After a short grace period, peers will broadcast new ARP requests which the VM will answer, and traffic start will flowing to the new Ethernet address. To speed things up, the VM may send out *gratuitous* ARP responses right away, to become reachable at the new address right away. This technique frequently reduces the adoption time of the new address from several seconds to a few milliseconds.

7.1.3 TCP Connection Migration

IP version 4 was designed before laptops became popular, and has limited support for mobility. The hierarchic name-space of 32-bit integers makes inter-network routing of packets relatively straight-forward, but also hinders mobility across subnets. If a host moves to a new subnet, it is expected to reconfigure its IP address, which causes trouble for connection-oriented protocols such as TCP. A TCP connection consists of an address and port number pair for each endpoint, and if one end of the connection moves to another address, packets will keep flowing to the old address for a while, until the lack of acknowledgements will cause one end to give up and abort the connection. Extensions to TCP have been proposed [Snoeren and Balakrishnan, 2000] that would allow one of the parties to re-initiate an existing connection, from a new IP address. After a host has moved to a new address, it sends a new SYN packet to the remote party, identifying the connection with a forgery-resistant *continuation token*. The other host verifies the integrity of the continuation token, updates the address information for the connection, and continues talking to the host at the new address.

The use of TCP connection migration would allow applications in a VM to relocate to a new address after the VM had migrated to a new host. Some of the problem with the approach is that it only work for TCP connections, and not other protocols such as UDP, and that it may require applications to be adapted in some cases.

7.1.4 Network Address Translation (NAT)

In cases where it is not desirable for each VM to have its own IP or MAC address, it is possible to multiplex the 16-bit port spaces provided by TCP and UDP instead. This solution is often used where multiple computers need to share a single routable IP address, such as when sharing a single Internet connection across a corporate LAN with many client PCs. Apart from providing more efficient utilization of the scarce IPv4 address space, this solution also has the benefit of preventing outside parties from opening connections to hosts hidden behind the router or firewall providing the NAT functionality, arguably increasing security.

All traffic in such a setup is routed through the NAT, and the NAT keeps state about each connection, *masquerading* outgoing packets to look as if they originate from the NAT box itself. The NAT indirects port numbers, so that an incoming reply may be appropriately forwarded to the client that opened the connection.

The downside to using NAT is that it loses the point-to-point properties of IP, because hosts behind the NAT cannot be contacted directly. One workaround is to explicitly open ports (*i.e.*, forward traffic on the port to a host behind the NAT) in the NAT, at the cost of having to perform manual configuration.

Because they reduce the problem of IPv4 address scarcity, and because they shield hosts behind the NAT from certain types of network attacks, NATs have become extremely popular and are deployed in many networks. The wide use of NAT has reduced the need for IPv6, and may ultimately hinder widespread adoption of that standard.

When utility computing systems move from university settings and into corporate and even home networks, the NAT problem becomes more apparent. Whereas a university may own large subnets of IP address space for historic reasons, this is rarely the case for later adopters such as smaller companies. As a result, an application-VM is likely to find itself subjected to several levels of network address translation, at both host and network boundary levels. A VM host environment may employ NAT in order to multiplex a single IP address, and that IP address may itself be a local address hidden behind a NAT'ing firewall.

Even when assuming that we can control most of the software stack on each node, we may be unable to exercise any control over the surrounding network environment, and any real-world deployment will ultimately have to deal with the NAT issue. For a utility computing system built using VMs, the main problems with NATs are the following:

Control of resources behind a NAT If the host is hidden behind a NAT, it becomes harder to remotely control it. Few existing systems deal with this issue, but assume that all hosts are either directly connected to the Internet, or assume the presence of a frontend-node through which other nodes can be contacted.

Node-to-Node Communication For workloads that require inter-node coordination or sharing of data, the existence of a NAT at the host level will be a hindrance, as will a network-level NAT when not all hosts share the same LAN.

Migration of NAT state When a VM migrates to another host, state about outgoing connections resides in the NAT from where it needs to be extracted. This is possible at the host level, where that NAT is likely to be under our control, but not at the network level.

7.2 Network Tunnels

One possible solution to the NAT problem is to *tunnel* all non-local traffic through a server with global, routable, IP address, so that only the tunnel connection needs to be restored after a move to a new network. Tunnelling means that traffic is encapsulated inside another protocol, for example by prepending an extra header in front of each packet. In addition a tunnel may provide VPN-like features such as traffic encryption. If the tunnel is implemented on top of a connection-less protocol such as UDP, the tunnel can be restored simply by sending out packets from the new location. One disadvantage to using a tunnel is the tunnels server may become a single point of failure, and a performance bottleneck. For performance, traffic to VMs on the local

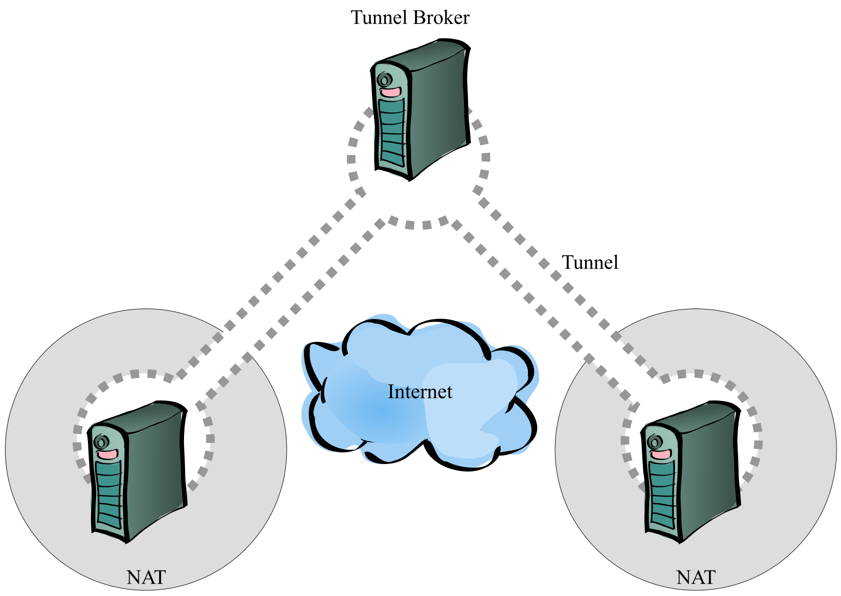


Figure 7.1: A tunnel that allows two NAT'ed nodes to connect through a third party.

network should be routed directly, and not through the tunnel. This can be done by maintaining the proper routing table entries in each VM.

A tunnel may be implemented at various layers of the protocol stack. For example, the SSH protocol includes the ability to tunnel single TCP connections, while other tunnel protocol such as IPSec [Kent and Atkinson, 1998] work at the IP level. At the link layer, *vnets* [Sundararaj and Dinda, 2004] is a tunneling mechanism for Ethernet frames, providing the illusion of a virtual Ethernet that spans multiple IP networks.

The purpose of most tunneling software is the creation of a Virtual Private Network (VPN) over an unsecured network. A key element here is the use of encryption, to prevent eavesdropping or forgery of network traffic. Apart from the cost of encapsulation, this adds a processing cost for every packet, and makes firewalling or traffic shaping based on packet headers impossible. Furthermore it requires a Public Key Infrastructure (PKI) or a similar key-distribution mechanism, to be in place. Because it frequently works at the OS level, encryption and key management support must be added to the OS kernel, adding to the size of the trusted computing base, and introducing new failure modes, *e.g.*, for mismatching keys.

In the scope of dealing with NATs, the primary benefit of having a tunnel is that once a single connection has been established through the NAT, it can carry any type of traffic, in an application-oblivious manner. Where TCP connection migration would require patching of all software to support cross-subnet migration, a single tunnel can accomplish the same for all applications. Another derived benefit could be the ability to annotate all messages with additional information, such as version numbers which would be useful, for example, for a distributed checkpointing algorithm. Figure 7.1 shows how two nodes behind NATs are able to connect through an external tunnel

Network configuration	MTU	TCP Bandwidth	Avg. RTT
Direct Gigabit Ethernet link	1500	808Mbit/s	117μs
Gigabit Ethernet link, through tunnel server	1472	385Mbit/s	370μs

Table 7.1: Bandwidth of an IP-in-UDP tunnelled TCP connection, and averaged roundtrip-times. The MTU is adjusted to make room for 28 bytes of extra IP and UDP headers.

Network configuration	Linux kernel compile
Master node directly connected to slave VMs	94.8s
Master node connected to slave VMs through UDP tunnel	96.4s

Table 7.2: Total build times for a parallel build of the Linux kernel, using direct network connections to two slave VMs on two hosts, and through IP-in-UDP tunnels.

server.

A generalized and failure-resilient tunnel implementation would function as an *overlay network*, and handle the routing of traffic between inter-autonomous networks, similar to what BGP routers do for the entire Internet today. As a starting point however, a simple star-topology implementation would provide the most important feature, namely the ability to communicate with and control NAT'ed VMs independently of their present location, though with limited scalability.

For our system, we have implemented a simple tunnel that encapsulates IP traffic inside UDP datagrams. In contrast to tunnelling through a TCP/IP connection, the use of UDP does not duplicate the reliability and congestion control features of the tunnelled traffic. In line with the end-to-end argument [Saltzer et al., 1984], our tunnel does not implement any encryption, as this can be handled at the application layer. The tunnel is implemented on top of the Linux TUN/TAP module, which acts as a forwarder between a virtual network interface and a file descriptor. This allows network packets to pass through a user space application, and arbitrary transformations to be applied. The tunnel server is a simple user-space application, running on a host with a routable IP address. The server routes traffic between peers using a table mapping tunnel IP addresses to real address and UDP port numbers. When a client behind a NAT sends a UDP packet to the tunnel server, the NAT keeps a UDP port open for a while, to allow a response to come back. To keep the port open at all times, the client has to continuously "ping" the server, to prevent the NAT closing from closing the port. Though there is no agreed standard setting for the NAT port timeout value, the value recommended by the IETF is five minutes [Audet and Jennings, 2007]. In practice we have found it necessary to ping the server twice a minute, but ideally the ping interval should be calibrated automatically.

Table 7.1 shows the performance overhead of tunneling, relative to that of a direct Gigabit Ethernet connection. We conjecture that most of the bandwidth loss and added round-trip time could be reclaimed by using an in-kernel rather than a user-space tunnel implementation. The loss of bandwidth is not always a problem; table 7.2 shows completion times for a parallel build of the Linux kernel, with three VM cooperating through the `distcc`[1] distributed compilation tool, with and without the tunnel. The slowdown caused by the tunnel in this case is less than 2%.

A final benefit to using a tunnel is that as long a VM has some kind of network access, a tunnel can be created. Tunnels do not have to be supported by the host environment, but can be configured entirely inside a customer's application VMs, yielding maximum flexibility, and preventing name-space collisions with other tunnels.

7.3 Access to Disk Storage

Many workloads benefit from having access on-disk storage. This is the case where the working set exceeds to available memory, or when wishing to store data permanently, surviving power failure or machine reset. Disks are also useful for storing checkpoints using the self-checkpointing algorithm. In this chapter we outline a number of alternative strategies for providing VMs with storage in an on-demand computing setup, that also supports VM migration.

7.3.1 Network Attached Storage

When migrating workloads between hosts, data stored on disk often creates a residual dependency problem. This is often solved by introducing Network Attached Storage (NAS), *e.g.*, through a network file system such as NFS [Sandberg et al., 1985] or AFS [Howard et al., 1988], or a remote disk access protocol such as iSCSI [Satran et al., 2004]. Provided that the file server or disk array remains accessible, the application is free to migrate. Like virtual machines, NAS systems provide a layer of indirection between the application and the physical disks, equipping a system administrator with the ability to alter a storage configuration without downtime, to migrate data sets between different storage arrays, perform backups, etc. Due to these abilities, enterprise data centers are likely to already employ networked storage, on which a virtual machine migration system can rely. Previously in this chapter, we described the various means by which a migrating VM can retain its active network connections, and when these techniques are observed, nothing special needs to be done to stay connected to network storage.

[1]http://distcc.samba.org

7.3.2 Storage for High Performance Computing

For high performance computing on commodity clusters, a NAS system cannot always be assumed available, both because of cost issues, and because a centralized NAS system is likely to become a bottleneck if simultaneously accessed by hundreds of cluster nodes. Depending of the workload and the configuration of the cluster, a number of possible solutions could be imagined:

No Storage Not all workloads need disk storage. A workload that processes its data set only once, without needing to store a large set of interim results, is able to read its input data via the network, and to return its output in the same manner when done. One example of such a workload is the SETI@Home search for radio signals from outer space, where a set of input data is downloaded, searched, and a simple yes/no answer returned. Such workloads do not benefit from having access to disk storage, and can instead be booted from a RAM-disk image.

Staging Some workloads process a data set that is too large to fit in memory, or make sparse accesses to the data set. In the first case, either the input data must be fetched from a server on demand, potentially causing the server to become a bottleneck, or the input data set can be staged to disk before the program runs. In the latter case, the data set may still fit in memory, but the memory can be utilized better by fetching needed parts of the input data on demand.

Read-only Disks When the data set or parts of it are identical for all instances of a computation, read-only disk images can be replicated along with migrating VMs. Read-only replication is simpler because all on-disk data can be sent ahead before a migration, or installed on all cluster nodes, without consistency issues. Another benefit is that read-only disk images can be compressed, *e.g.*, using SquashFS[2]. Directories that must be writable, *e.g.*, `/tmp` or `/var/log`, can be mounted on a RAM disk, or a *stackable* file system such as UnionFS [Wright et al., 2006] can be used to give applications the issusion of a writable file system. Data on RAM disks or in a memory-backed writable overlay will be automatically transferred as part of a OS migration.

7.3.3 Partial File Access

Systems that rely on staging or fully pre-fetched read-only disks, may suffer from long startup times, especially if all job instances fetch their input data from a single, centralized file server. In chapter 8 we show how the use of a hypercube network broadcast can alleviate this problem, but this solution still suffers from longer startup-times than necessary, due to having to wait for the entire data-set before work can begin. A solution to this problem is a network file system that allows for partial file access, so that the disk blocks that are needed first can be fetched before those that are needed later

[2]http://squashfs.sourceforge.net

on, or not needed at all. A NAS system such as NFS or iSCSI would allow this, but for a large number of clients the server becomes a bottleneck. A peer-to-peer file system such as Shark [Annapureddy et al., 2005] would provide a scalable solution, where a large number of clients need partial, demand access to a shared file system.

7.3.4 Software RAID

Another option that we have been investigating, is using the built-in software RAID-1 and iSCSI functionality of the guest Linux to implement disk mirroring before and during OS migration. We imagine that RAID-5 could be used in cases where data on disks would require a higher level of fault-tolerance. Multiple hosts can act as storage targets for one another, increasing availability at the cost of some network traffic. The main disadvantage of this solution is that software iSCSI target must be running on a destination node beforehand.

In one experiment, we configured iSCSI software targets to run in the trusted portion of each of two nodes, and managed to self-migrate a running kernel with an active, writable disk, between the nodes, with negligible downtime, and controlled by a simple shell-script inside the migrating VM.

1. Initially, the guest OS is configured to run a software RAID-1 mirror, but with only one local (primary) disk attached. Due to the lack of a spare disk, the mirror functions exactly as would a single disk.

2. Before migration, an identically sized virtual disk is created at the destination node, and the OS connects to this disk, *e.g.*, using iSCSI. When the disk is connected, it is added to the RAID-1 array.

3. The software RAID-1 implementation inside the guest Linux discovers the new (secondary) disk, and starts synchronizing data from the currently active disk to the newly added replica.

4. Once the replica is completely synchronized, the RAID-1 array continues in mirroring mode. Every write to the array is replicated to both disks, and reads may be serviced from any one of the disks, to improve bandwidth.

5. At this point, the OS itself may be self-migrated to the destination node. During migration, the RAID-1 code keeps both disks synchronized.

6. After arrival of the main-memory checkpoint at the destination node, the primary disk is no longer needed, and can be disconnected. The RAID-1 implementation will notice that the primary disk is gone, and fail over to exclusive use of the secondary one.

Thus, modulo the problem of having to add an iSCSI software target to the trusted installation, live disk migration in this setup is trivial. The only problem we experienced was that when disconnecting the primary disk after migration, the Linux iSCSI driver was reluctant to give up the disk, and instead resorted to a long time-out, in the hope that the network would recover. During this grace period, all disk writes would be delayed, causing applications to hang. The iSCSI driver was unaware that the disk was mirrored, and considered the connection loss to be a disaster to be avoided at all cost. Though we expect that it would have been easy to fix, it is interesting to note how this combination of different subsystems leads to unexpected misbehavior.

7.3.5 Migration of On-Disk Data

The final, and most comprehensive solution, is to include on-disk data in the state that is migrated. Just as memory pages are sent and possible re-sent during migration, the same can be done for disk blocks. During migration, blocks are marked as dirty when written to, and re-transmitted until a consistent copy has been transferred. Workloads where a large data set is read and written repeatedly will benefit, because this solution avoids the need for a centralized NAS server, and does not leave residual dependencies on the original host. Finally, it does not depend on subsystems or protocols such as iSCSI, and does not inflate the trusted install with a NAS server.

We have prototyped disk-migration for read-only disks. The `cstrap` in-VM firmware (described in chapter 8) was extended with a disk driver, that we also use when resuming checkpoints from disk. Before the checkpoint of memory is sent, the user-space checkpointer process reads disk blocks from one or more block devices, and the checkpoint loader on the receiving side writes them to the local disk there. This simple technique works as long as the disk is read-only, but does not track changes to the disk during migration, and will result in inconsistent disk contents if the disk is writable. What is currently missing is a facility for tracking these changes, so that changed disk blocks can be retransmitted, just as for main-memory migration.

7.3.6 Checkpointing to Disk

The self-migration algorithm is able to incrementally checkpoint an entire operating system, while it is running. With this functionality, the operating system essentially becomes a persistent system. The advantages of persistent systems are that they retain their state through power outages, and that they free application writers from implementing persistence features in their own programs. Simple self-persistence is already implemented in popular desktop-operating system such as Windows and Linux, in the form of "hibernation" or "suspend-to-disk" features. However, these simple techniques suffer from very long freeze times, and are not suitable for interactive systems.

Because of the split design, where connection setup and data transfer are handled by a user-space program, turning network self-migration into disk self-checkpointing is trivial. Instead of writing to a socket, the checkpoint data is written to disk. The format is slightly different, due to the fact that the disk is randomly addressable. The disk checkpointer opens a raw block device, with the `O_DIRECT` flag that tells Linux to bypass the buffer cache. It first writes the checkpoint loader, and then the pages read from the checkpoint device node. Non-consecutive page numbers are addressed using the `seek` system call, to result in a 1:1 correspondence between disk and memory layout of the checkpoint. This means that the checkpoint can be restored quickly and simply by reading it back into memory, and performing the recovery steps described in section 6.5.

7.3.7 Virtual Machine Storage Systems

Special storage systems for VM systems have been developed. While it is fairly simple to provision a virtual disk as an extent on a physical disk, time and space can often be saved using optimizations such as copy-on-write disks. Parallax [Warfield et al., 2005] is an example of a storage backend that provides such functionality, and is simple to extend for other uses.

In certain scenarios where virtual disks may be too coarse-grained, virtual machine systems could benefit from access to a higher-level, versioned shared file system. A system such as Ventana [Pfaff et al., 2006] provides a network file system-like service, but with support for branching of sub-trees, integrated with VM checkpoints. Like many recent network file systems, Ventana builds on object storage, separating meta-data and object data in separate storage systems. File systems are exported from Ventana through an NFS interface, making it compatible with most commodity guest operating systems.

7.4 Chapter Summary

Virtual machines require network access, to allow coordination with other VMs, and to communicate with end-users, and access to disk storage for input and output data. It is not always acceptable for a VM to be bound to a certain network interface or IP address, and several ways for relocating a VM's network connections exist, at different layers of the protocol stack. In cases where it is not desirable to outfit each VM with a unique MAC or IP address, network address translation can be used to multiplex a single IP address through sharing of the TCP and UDP port spaces. Though NAT often prevents incoming connections, a possible workaround is to tunnel traffic through a virtual private network (VPN). This solution can be implemented inside application VMs, without growing the trusted installation on host nodes.

For storage, we have outlined a number of solutions that will scale to a large number of VMs, while still allowing VMs to migrate between hosts. Common to all of them is that they build on existing technologies, and can be implemented without growing the TCB on cluster nodes.

CHAPTER 8

Network Control Planes

Virtual machine systems described in literature focus mostly on the low-level issues of tricking multiple guest operating systems into sharing a single physical machine, and the many benefits to having a layer of indirection between the operating system and the bare hardware with regards to security and server utilization. In contrast, less effort has gone into securing the *network control plane* software that is a necessary part of most VMM systems, even though a design mistake at this level could adversely affect the integrity of all VMs in a system.

Popular systems such as VMWare [Sugerman et al., 2001] and Xen [Barham et al., 2003] provide feature rich network control planes for remote administration. Unfortunately, complex control software may well be an Achilles Heel to the security provided by the VMM, as complexity often leads to bugs and vulnerabilities. Control plane software needs ultimate power over guest VM instances, so there is a danger that a single exploitable programming error in the control plane may jeopardize the integrity of all hosted VMs. In this chapter we describe the design and implementation of two simple network management models for on-demand computing. Under these models, cluster nodes can make their resources available to the network, without resorting to the complexity of traditional VMM control planes.

8.1 Control Plane Requirements

The purpose of a virtual machine monitor is to host one or more guest virtual machines. The VMM itself is necessary for safe multiplexing of physical resources such as CPU cycles, memory pages, and disk blocks. Because its functionality is so limited, the VMM is ideally a very simple piece of software. Lowell et al. [2004] describe a purpose-specific "Microvisor" VMM implemented entirely as Alpha PAL-code, and the original L4 kernel consisted of only 12 kilobytes of binary code [Liedtke, 1996].

To arbitrate access to the multiplexed resources, the VMM needs to be configured with information about who has access to what, and what workload should be running in each VM. For some deployments, it is conceivable that all of this information could be hard-coded in the VMM, but most systems will need more flexibility, and should be able to obtain configuration information at run-time. Obviously, the integrity of this information is important, as a wrongly configured VMM will fail to provide any safety at all. In other words, virtual machine systems have a boot-strapping problem: Once a set of VMs are configured and running, the system is safe and simple, but getting to this point of "virtual nirvana" may require a sacrifice of those same virtues.

In a data center or cluster-computing scenario, the simplest way to obtain configuration and workload information will be through the network, and it is therefore tempting to expand the VMM with a stack of network protocols such as TCP/IP. On top of these comes some form of remote login system, or an RPC [Birrell and Nelson, 1984] implementation, for remote management. If the system is expected to provide features such as live VM migration, then a service that can read or write VM checkpoints has to be provided as well. To prevent against network attacks, the migration service and the general management RPC service must be protected by as uite of encryption protocols. Because several clients can be expected to connect simultaneously, threading or another form of concurrency must also be supported.

At this point, it is tempting to simply add an entire commodity operating system to the trusted install. Compared to a custom solution, many programmers will find a commodity host operating system simpler to develop for, and the compatibility with a commodity OS will allow new users to experiment, without having dedicated hardware available. The host OS can also provide device driver abstractions, as described in chapter 4. At the extreme, the hypervisor and host OS can be merged into one, as is the case with the User Mode Linux[1] and KVM[2] projects, where the Linux kernel is used as a VMM.

From a security point of view, the addition of a host OS is problematic, because it greatly inflates the size of the Trusted Computing Base. Empirical studies [Shen et al., 1985] indicate that there is a correspondence between software complexity and the risk of errors. In settings where stability and security are of paramount importance, the use of a large and complex host OS for the control plane should therefore be avoided.

8.2 Laundromat Token System

Our original vision for a network management control plane was inspired by a simple real-life example: Laundromats are an example of a machine-provided service that is made accessible to untrusted customers who pay pay with laundromat tokens, without any initial negotiation phase, and without long-term commitment by any of the parties. If clusters were more like laundromats, untrusted customers would be able to buy access to their resources very easily, and cluster owners would be able to make money from their use, without wasting too much time on legalities.

The laundromat model is interesting when selling access to computational services, because it does not require the parties to enter into any formal agreement before work can begin. The value of each token is limited, and so the initial investment to start a

[1]http://user-mode-linux.sourceforge.net/
[2]http://kvm.qumranet.com/

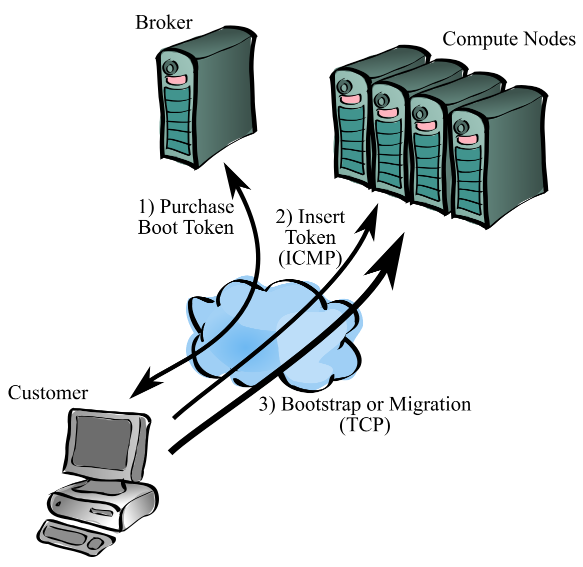


Figure 8.1: Creating a new VM with a freshly purchased Boot Token, containing s_n. After the VM has been created with with a boot token in an ICMP payload, bootstrap or migration takes place over a regular TCP connection.

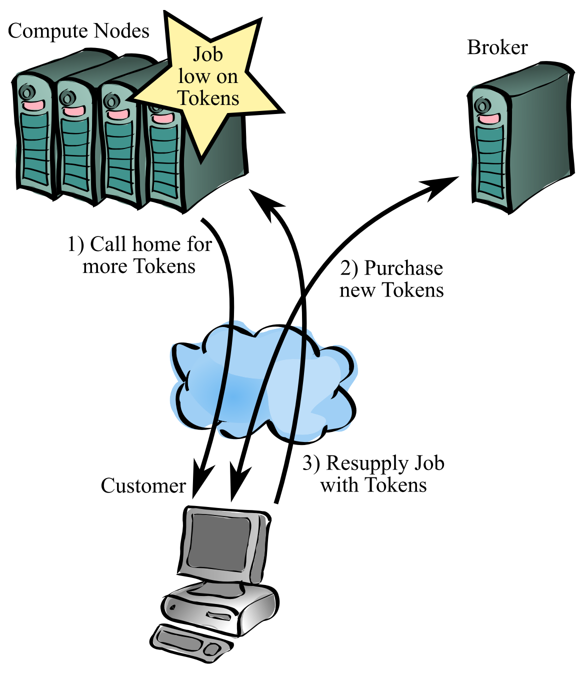


Figure 8.2: Resupplying a VM that is about to run out of tokens, with new element of token chain s_i. The VM is responsible for maintaining a positive token balance.

new job in a VM is small. If the job makes the expected progress, more tokens can be purchased to keep it running. If the vendor fails to provide service, *e.g.*, the job stops responding or runs too slowly, the customer can move the job to a competing vendor. Individual machines need to know very little about payment: all they need to know is the number of tokens required to allow access to their resources. The actual payment plan is decoupled from the usage of the tokens; the Laundromat owner is free to use any kind of token sales or to change the policy for token sales at any time.

Laundromat tokens are similar to *leases*, a popular way of specifying time-limited reservations in distributed systems. A lease describes the right by the lease-holder to use a certain resource until the lease expires. The lease can typically be extended, and the point of using a lease, rather than, *e.g.*, a lock, is that a crashed or disconnected node is prevented from holding on to a resource forever. When dealing with untrusted clients, leases should be designed in a way that prevents forgery, to prevent unauthorized reservation and use of resources. A common way of doing this is by signing leases using an encryption key that is universally trusted. A simpler and faster method is to sign only the first lease for a certain resource, and use a one-way function to impose an ordering on all subsequent extension to that lease. The *Evil Man* token system is designed using the latter method, in order to be as simple as possible to implement on the nodes.

The Evil Man token system builds on a so-called one-way hash chain that is hard for an attacker to reverse, but easy for the system to verify. Tokens have a fixed, low value, so that the system does not have to deal with making change, and tokens are minted for and only valid on a single node, to prevent them from being spent concurrently at two different locations.

To submit a job to a node, a customer first obtains one or more tokens from a brokering service. The customer pays the broker using a commodity payment system and gets back a boot token and a sequence of payment tokens. A token is merely a one-time password. To efficiently generate a sequence of one-time passwords we use a one-way hash function f. We start by randomly generating a single initial one-time password s_0, and derive the next password s_1 as $f(s_0)$. In general, the n^{th} password is derived from s_0 as $f^n(s_0)$. Initially the broker establishes a sequence of, say, one million passwords ($n = 10^6$). To sell a customer the first 10 tokens, the broker tells the customer the value of s_{n-9}. These are the first tokens purchased from the chain, so the broker inserts s_n and s_{n-9} into a newly created boot token which it then signs using a secret shared with the node. The customer then sends the boot token to the node. The node verifies boot token signature, and now knows to trust s_n as the end of a valid token hash chain. It also checks that $f^{10}(s_{n-9}) = s_n$, and if so increments the customers local token balance by 10. If the customer later wishes to purchase 10 additional tokens, the broker tells the customer s_{n-19}. The customer can now send s_{n-19} to the node, which is able to verify that $f^{10}(s_{n-19}) = s_{n-9}$. If s_0 is ever reached, the broker can generate a new chain and supply the customer with a new boot token.

Tokens are consumed at a rate proportional to the customer VM's use of resources.

When a customer VM reaches a negative token balance, the privileged control software on the hosting Evil Man node will respond by purging the VM from the system. To stay alive, the customer VM is responsible for keeping a positive token-balance, by purchasing tokens from the broker, either directly or by calling on a home node to do so on its behalf. When using the latter method, the customer VM does not have to carry valuable access credentials around, and the home node will be able to monitor the progress of the VM before making further investments. Before running out of tokens, the VM is given a chance to suspend itself to disk and shut down, thereby reducing its token consumption rate dramatically. During the remaining grace period, the customer is able to revive a suspended VM by purchasing more tokens from the broker and sending them to the node. Figure 8.2, shows a customer VM calling home for an additional token supply.

Upon job completion, it is possible for the job to request a refund of any remaining tokens on the node. The node returns a signed redemption certificate that can be presented to the broker for a refund.

Double-spending of tokens is prevented by having separate hash-chains for each job and each node. Because tokens are short-lived, there is no need for a revocation mechanism, as would be necessary in a certificate-based system. Figure 8.1 shows a customer purchasing an initial boot token from the broker and with that submitting a job to a compute node. The token purchase and refund protocol is intentionally left unspecified, so that site-specific policies can be implemented.

8.2.1 Self-Inflation Network Protocol

In chapter 5, we saw how the self-migration technique allows an unprivileged VM to checkpoint itself onto another node, and this obviates the need for migration functionality in the privileged parts of our system software. However, the problem remains of receiving and resuming the checkpoint in a new VM at the destination node, without having having a receive-capable TCP server with the ability to create new virtual machines as part of the trusted software base there.

A related problem is the case of wishing to bootstrap a new guest VM onto an Evil Man node. In this case, we need the ability to authenticate and receive a guest VM kernel image sent via the network, and we need a way of deciding what limits to impose on its use of resources.

Our proposed solution to these problems is a technique that we call *self-inflation*. Self-inflation involves a separate and very simplistic network protocol for requesting the creation of a new VM on a remote node. This VM contains a small and unprivileged bootloader to which a kernel image can be sent for boot-strapping, or which can be used as the target of a migration. Figures 8.3 and 8.4 show bootstrapping and self-migration using the self-inflation protocol. In detail, the protocol has four stages:

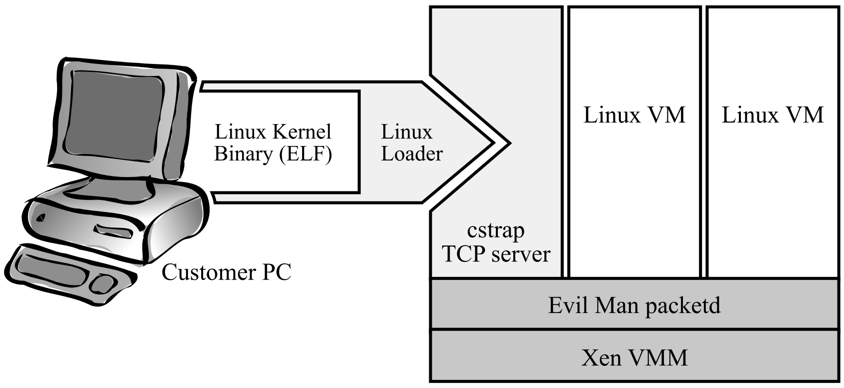


Figure 8.3: Evil Man bootstrapping mechanism. The customer injects a loader executable into an unprivileged `cstrap` boot server, running in a VM. The loader takes control of the VM, and bootstraps the guest OS.

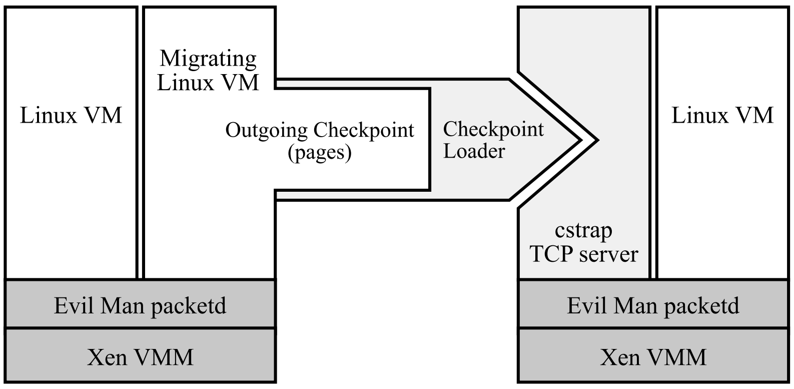


Figure 8.4: Self-Migration of a VM in Evil Man. A checkpoint loader executable is injected into the `cstrap` VM, followed by the actual checkpoint data. The loader receives the checkpoint data, and restores the checkpoint into a running guest OS.

1. The client (for example software running on the customer's PC or customer software running on another Evil Man node) first needs to find out the physical address of the node from it's IP address. In an Ethernet, this happens as in normal IP by broadcasting an Address Resolution Protocol Request (ARP Request) to the local segment. On the client side, this *address resolution* step is normally performed by the TCP/IP stack. On the node, there is no TCP/IP stack, but Evil Man nodes understand ARP Requests, and the node configured with the requested IP address will respond with its Ethernet MAC address in an ARP Reply packet. For convenience and RFC-compliance, normal ICMP Echo Requests are also responded to with the appropriate Echo Reply.

2. When the Ethernet address is known, the client requests the creation of a new VM by sending a special IP Internet Control Message Protocol (ICMP) packet to the node. The ICMP payload carries a *boot token*, a binary structure describing the access permissions given to the new VM, and containing the initial tokens provided by the customer as payment, as described in section 8.2. The boot token is HMAC-signed [Bellare et al., 1996] using a previously established shared secret, and upon successful verification of this signature, the new VM is created. Inside the new VM a small TCP server (derived from UIP [Dunkels, 2003]) is started.

3. The client connects to the TCP server inside the new VM, and uploads a binary executable (the *loader*) which fits inside the address space of the TCP server. After the loader has been successfully uploaded, and its checksum compared to the one specified in the boot token (to prevent a man-in-the-middle from hi-jacking the new VM by tampering with the loader), the loader is given control of the TCP connection.

4. Finally, bootstrap or migration of the incoming VM can take place. The loader is responsible for the rest of the transfer, for example, for decoding an incoming kernel image and creating its virtual address space, or for receiving an incoming migration-checkpoint and restoring it to running state.

8.2.2 Implementation Complexity

Evil Man runs on top of the Xen virtual machine monitor, using a modified version of XenLinux 2.6.16 as the example guest OS, as well as the custom-developed *cstrap* boot-loader with TCP and HTTP support, provided by UIP Dunkels [2003]. To turn XenLinux 2.6.16 into a self-migrating OS, we had to add or change 2,100 lines of code. Self-migration is performed entirely without VMM involvement, so we did not have to make to the Xen VMM.

Apart from the self-migration functionality inside XenLinux, Evil Man consists of a privileged network server, called `packetd`, and a specialized VM operating system, called `cstrap`. The job of `packetd` is to listen on the network for incoming ARP and ICMP packets and respond to them, and, in case the ICMP carries a payload with

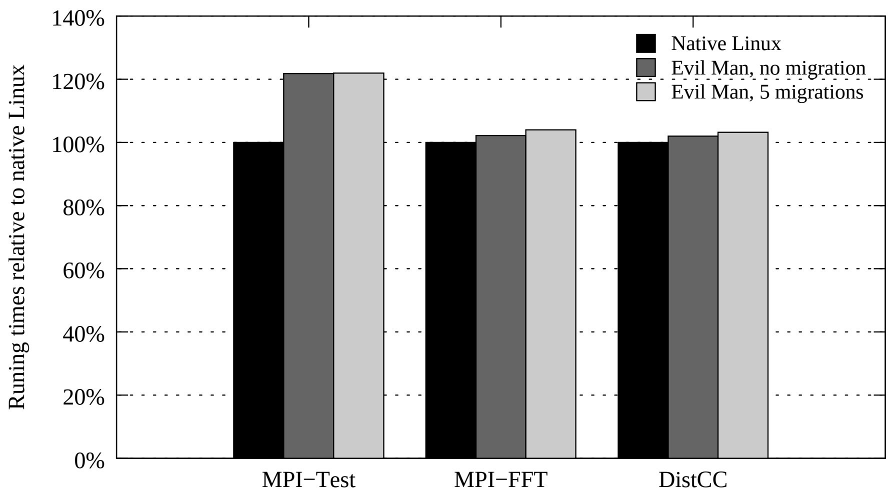


Figure 8.5: Relative run-times of Evil Man vs. Native Linux 2.4.

a valid boot token, instantiate a new VM. We use a simple keyed hash function for message authentication, as this allows us to reuse the hashing code that is also used for the token system. The main loop of `packetd` is currently 145 lines of C-code, excluding the secure hash function (currently SHA-1) which is 200 lines of code. Apart from the network driver, this is the *only* privileged network-facing code in Evil Man.

8.2.3 Performance of the Evil Man System

We have been evaluating the usefulness of our system on a small cluster of commodity PCs, and have made basic measurements of the performance of our system which we present in the following:

We constructed three benchmarks that mimic what we expect to be typical uses of the system. One test is a parallel build of a large C-code base (the Linux 2.6 kernel) using the `distcc` distributed compilation tool, and the other two are examples of scientific computations (a parallel Fast Fourier Transform (FFT), and an MPI test suite) implemented on top of LAM-MPI [Burns et al., 1994].

We run our tests on a cluster of Dell Optiplex 2GHz Intel Pentium4 (without Hyperthreading) machines, connected via Intel E1000 Gigabit Ethernet adapters. Eleven nodes from the cluster are allocated for the tests, with eight of the machines being active at the same time, and two kept as spares to use as migration-targets. The last node is acting as Master, submitting jobs to the other Slave nodes. All nodes are running the Evil Man `packetd` server inside Xen's Domain-0 VM. Before each test, Evil Man VMs are bootstrapped with a XenLinux 2.4.26 guest kernel. In the Native Linux case, the machines are running a native 2.4.26 kernel with configuration and optimization settings similar to the guest kernels running under Xen.

At the start of each job, the job control script running at the Master node bootstrap a new VM onto each of the slave. The Master also acts as an NFS server, from which the slave VMs get their root file systems. In all cases each job instance is allowed the use of 200MBs of memory. Each of the tests is run in three scenarios; under native Linux directly on the hardware, in a VM under Evil Man, and in a VM under Evil Man while performing a number (4 for `distcc`, 5 in the MPI tests) of randomly selected live migrations during each run.

The graph labeled DistCC in Figure 8.5 shows Evil Man performance relative to native Linux. Evil Man nodes take 2.0% longer to complete without migration, and 3.2% longer when four random migrations take place during the test. This test is mostly CPU-bound, suffering only a slight slowdown from the use of Evil Man on Xen. The MPI-FFT graph shows relative completion times for the MPI FFT test. In this test, native Linux also comes in ahead, with Evil Man jobs taking 2.2% (resp. 4.0% with five migrations) longer to complete on average, which we attribute to the I/O processing overhead of the Xen VMM. The final test, the LAM-MPI test suite, is mostly an I/O bound test, measuring network latencies and communication performance of an MPI setup. Compared to the other tests, Evil Man fares less well here, with a 21.8% (21.9% with 5 migrations) overhead compared to native Linux. We expect that most of this overhead is due to the way Xen tries to reduce switches between VMM and guest VMs by batching multiple network packets, at the cost of adding latency.

From the figures we can conclude that with the possible exception of very I/O-bound applications, the overhead of using Evil Man is not prohibitive. The extra startup time for the nodes is offset by the added flexibility and security, and, in practice, will be completely amortized for longer running jobs. The option of performing live migration of jobs, in response to load imbalances, to deal with failing hardware, or to preempt a job to let another one run, also comes at a very low cost.

8.3 Second-Generation Control Plane

The goal of our initial "Laundromat" design was to come up with an absolutely minimal network control plane for our cluster nodes. With a control plane implementation of only a few hundred lines of C, we have come close to reaching that goal, even when adding the complexity of the VMM and device drivers to the total weight of the implementation. Our design was based on having a VMM available which included device drivers in the base software, *i.e.*, the original version of Xen, as described by Barham et al. [2003]. Unfortunately, the Xen community was unable to bear the burden of maintaining a complete set of drivers for Xen, and opted instead for a model where drivers reside inside a privileged VM, running Linux. The trade-offs involved in the design choice are described in section 4.3, but for our purpose of control plane simplicity it meant that in practice we were left with little choice other than to add a complete Linux kernel to our base installation.

Another problem with the original design was that it did not work well across closed subnets. The initial boot token is carried in a “magic ping” ICMP packet, that can only be received if the node has a global IP address, or the sender is on the local subnet—though possibly represented by some form of proxy service. As described in section 7.1.4, this is not only a problem for our system, but indeed for any system that wants to remote-control nodes on closed subnets.

The above two problems let us to conclude that we had to partially redesign the network control plane. We were unwilling to move our implementation to another VMM (such as VMWare’s ESX server) which did not require a full Linux kernel to run, and while still wanting to keep the design simple, we felt that the presence of a fully-functional OS in the privileged VM could be exploited without compromising safety. Separate work on the Pulse system, which we shall describe shortly, had led us to discover various techniques that could be used for controlling nodes behind NATs in secure and scalable ways, and in combination with extensions to our bootstrapping implementation, we arrived at a new control plane design. The second generation control plane, known as “Evil Twin”, differs from the first in the following ways:

first in the following ways:

Everything is HTTP All connection setup is handled using HTTP. This allows for integration with the wast body of preexisting web server, database, and scripting software. Our `cstrap` bootstrapper is extended with the ability boot over an HTTP connection, and each VM runs an HTTP server that can act as a boot server for other VMs.

Pull model The control plane and the VM bootstrapper never accept incoming connections. VMs boot from an HTTP URI, and migration and VM forking happens by actively opening a HTTP connection to the source VM. This design choice improves security by not requiring that privileged services accept connections, and helps with NAT penetration.

Caching design The workload on each node is described in a single data file, replicated from a control node. Changes to the workload are announced using a cache invalidation protocol, and the file gets fetched using HTTP. The use of the Pulse cache-invalidation mechanism allows us to control nodes behind firewalls and NATs. The benefit of this model is that all state on nodes is soft-state, that is cached for efficiency, but also can be recreated if it is lost.

IP tunneling To work across NAT’ed subnets, network traffic may be optionally routed through IP-in-IP tunnels. The use of tunnels allows a set of application VMs to form a private IP address space, and eases communications between VMs on NAT-hidden nodes.

Evil Twin relies on a centralized web service for workload information, and on a wide-area cache invalidation mechanism, which we are going to describe next, for receiving workload update notifications.

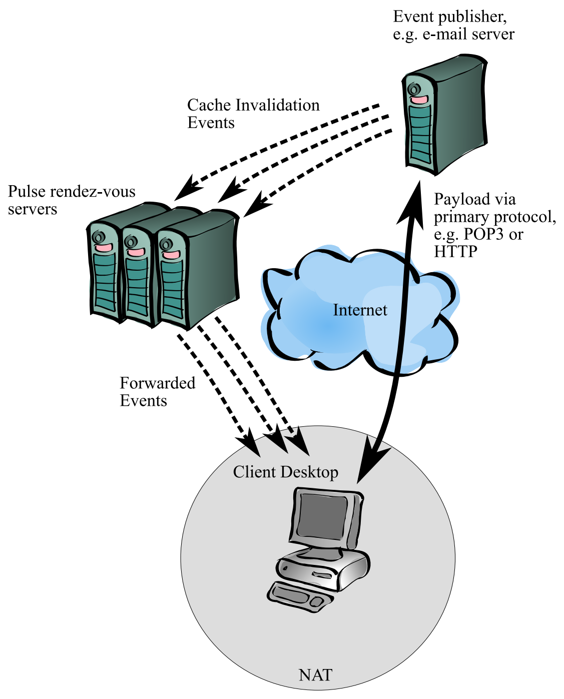


Figure 8.6: Flow of Events and Payload data in the Pulse system.

8.3.1 Secure Cache-Invalidation with Pulse

End-user computer systems such as laptops, desktops, handhelds and smartphones, increasingly cache replicas of remote information. Mail readers cache remote mailboxes, web and RSS readers cache news summaries, and operating systems cache system files that are developed and compiled remotely. Some of this cached information stays unchanged virtually forever, and other information is updated very frequently. To stay updated, client software often relies on polling the master replica at certain intervals. Unfortunately there is no formula for deciding what the correct interval is for a given replica, leading to a classic tradeoff between update latency and bandwidth use. Recent work has explored the use of numeric optimization of polling intervals, so that polling frequency is adapted to better follow update rates of various event sources. In this way, latency is reduced but not eliminated.

Previous work on distributed file systems has show that for read-mostly loads such as the ones described, the use of callback-invalidation leads to better consistency and reduces the loader on the master replica. Systems such as AFS, NFSv4, and CIFS all support the use of callback-invalidation, provided a callback-path exists between server and client. This is not frequently the case the case for end-user systems, and therefore polling remains the only viable strategy.

To avoid polling, several event systems have been proposed for wide-area use, *e.g.*, Scribe [Castro et al., 2002] and Herald [Cabrera et al., 2001]. Because subscribers may have intermittent connectivity or may be unable to accept incoming traffic, systems such as Herald employ rendezvous servers which store and forward events on behalf of subscribers. The problem with the use of rendezvous servers, is that they have to enforce some policy with regards to how much event payload data they are willing to store, or risk falling victim to DoS attacks. This policy in turn limits the types of protocols the event system can accommodate. Furthermore, forwarding information through untrusted intermediaries may be problematic for reasons of privacy, integrity and confidentiality.

Pulse, our new system, is not a generalized publish-subscribe system or multicast overlay, but rather an event notification mechanism dedicated solely to the problem of cache invalidation. Pulse is built on the assumption that the master (server/publisher) and the replica (client/subscriber) already possess a reliable communications channel (such as HTTPS) for fetching updates, but that maintaining a connection through this channel at all times is too costly or impractical. E.g. a constantly open TCP connection will be costly in terms of server-side memory, and impractical because it prevents host mobility and server-side fail-over. The purpose of Pulse is not to replace the primary communications channel, but only to notify the client *when* its cache should be updated. When this notification reaches the client, the client is able to verify its authenticity, and then flush its cache or update its contents using the primary channel right away. The main contribution of Pulse is how it decouples event notification and event payload propagation, and how this decoupling makes it policy-free and compatible with existing protocols. In addition, Pulse is designed in a way that preserves whatever confidentiality and privacy guarantees the primary channel protocol provides, even when notifications have to be forwarded through one or more rendezvous intermediaries.

The central idea in Pulse is the use of a one-way function $H(x)$ for the verification of event authenticity. An event is identified by two integers s and $H(s)$. When the master replica publishes an update, a new secret integer s is randomly generated and stored at the master. The derived value $H(s)$ is given to the subscriber, piggy-backed onto the primary protocol, *e.g.*, as an extra HTTP header field. In addition, the addresses of a number of rendezvous servers are provided, so that the client knows where to ask for update notifications. After synchronization, the client knows $H(s)$, the identifier of the next event from the master, *e.g.*, when the mailbox or web page is updated. The subscriber also knows that because s is so far secret, and because $H(x)$ is a one-way function, the only way to ever learn s is if released by the publisher. The subscriber may now subscribe to the event identified by $H(s)$, at one or more rendezvous servers. When the state of the master replica changes, the master can publish an invalidation-callback, simply by releasing s to the rendezvous servers. The rendezvous servers, and eventually the client, can now trivially verify event authenticity by simple application of H. Because H is a cryptographically strong one way function, events cannot be forged, and because s is randomly chosen, nothing can be learned by eavesdropping on the event channel that could not have been learned from the primary channel, preserving security. The system scales well because many events can fit into memory at each rendezvous server, and because an unlimited number of rendezvous servers may

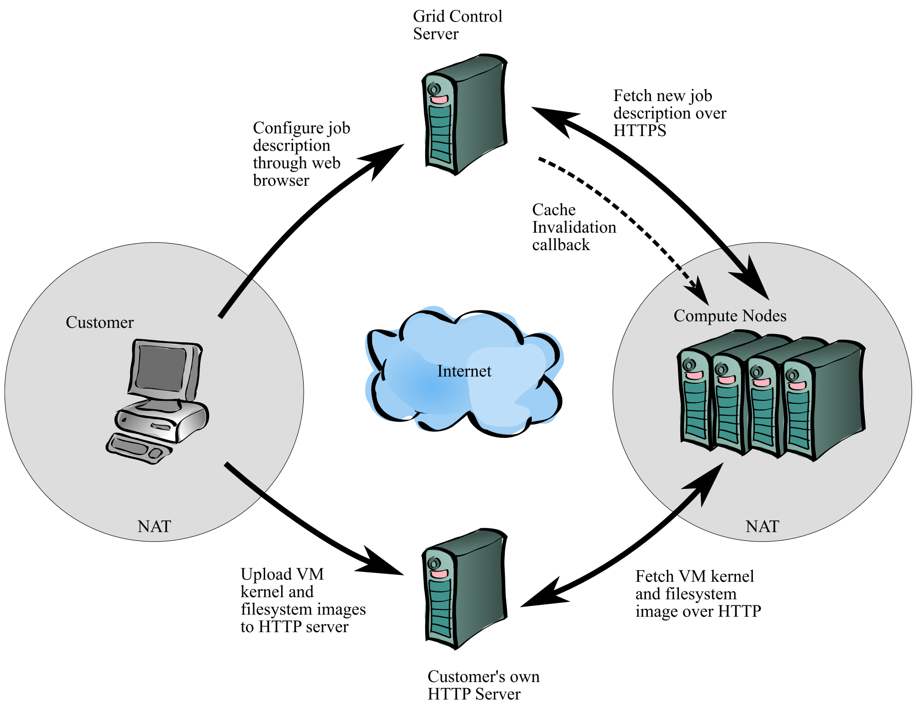

Grid Control Server
Configure job description through web browser
Fetch new job description over HTTPS
Cache Invalidation callback
Customer
Internet
Compute Nodes
NAT
NAT
Upload VM kernel and filesystem images to HTTP server
Fetch VM kernel and filesystem image over HTTP
Customer's own HTTP Server

Figure 8.7: Instantiation of a Grid job in the second generation, caching model.

be deployed to support many clients.

In practice, Pulse runs over DNS-formatted UDP packets, which means that Pulse traffic is able to traverse most NATs and firewalls. When a client is behind a NAT, it has to renew its event subscriptions a regular intervals, between 30 seconds and five minutes depending on the particular NAT configuration. The difference between this "keep-alive RPC" Burrows [2006] method and traditional polling is that notifications are instantaneous, rather than being dependent on the polling interval.

In summary, Pulse is system for callback cache-invalidation in the wide area. The decoupling of event notifications and payloads makes Pulse protocol-neutral and policy-free, and preserves privacy, integrity, and confidentiality.

8.3.2 Caching Control Model

Central to our second generation control plane is that the workload description for each node is maintained as a cached replica of a centrally controlled master, and that all distributed resources are named using HTTP URIs. The use of HTTP means that centralized job control sofware can be written using popular scripting languages, and that existing security mechanisms for HTTP can be used for creating a mutually authenticated communications context between customer, job control server, and the compute nodes.

The second model utilizes the presence of a privileged Linux VM as part of the trusted and privileged installation on each node. We conjecture that if this software never accepts incoming connections, and only talks to a trusted control server, it will still be safe from compromise. Naturally, this division of concerns centralizes control at one or more servers, and thus requires that these are implemented and administered with strong attention to security.

Each node in the new model contains a description of its entire workload. The description is a list of all the VMs running on the machine, *i.e.*, information about maximum memory allowance, access to disk ranges, and maximum CPU load. The description of each VM also contains a *boot URI*, the location of the VM bootloader, kernel image and additional boot files, bundled up as a single binary object. When the VM boots, it contains a small firmware, similar to mstrap described previously. This firmware contains Xen network driver and disk drivers, and a simple HTTP client. The firmware is started with the boot URI as argument, and the HTTP client proceeds to fetch the kernel bundle from this URI. The bundle is headed by a boot-loader, and as in the Laundromat system, this means that different kernel formats, and checkpoints, can be booted.

If the workload description for a node changes, a cache-invalidation callback is sent from the control server to the node, via the Pulse mechanism. Pulse events preserve the security properties of the primary communications channel, in this case HTTPS. When the integrity of the Pulse cache invalidation callback has been verified, the node control plane software updates its workload description, and creates or destroys VMs accordingly.

Because Pulse events are able to penetrate NATs, this design allows nodes to reside on firewalled subnets. The trusted installation on each node is larger than in our original design, but all control data is pulled from a trusted source, and no incoming connections accepted, so the potential for exploiting the control plane is still limited. Figure 8.7 shows the flow of information under the new model. Figure 8.8 lists a sample workload description, with a single VM description instantiated at two different hosts.

8.4 Hypercube Broadcast of VMs

When multiple nodes are on the same network, but need to obtain the workload data from either a single local server, or from a remote server, the local server or the remote link will become a traffic bottleneck. A large part of the workload will be identical across all of the nodes, and bandwidth can be saved by utilizing a multi-cast tree or a similar local peer-to-peer distribution method. One method that we found particularly interesting is the use of a hypercube-broadcast of VM images, to be able to boot a large number in a scalable way. The `cstrap` in-VM firmware supports both active (HTTP client) and passive (TCP server) opening of TCP/IP connections, and using this functionality we were able to implement hypercube broadcast of VM images.

```
<?xml version="1.0"?>
<workload>
 <vm id="1" version="67" maxmem="256"
    booturl="http://192.38.109.131/evilman/pack.cgi"
    cmdline="init=/linuxrc root=/dev/ram">
  <vbd size="60" virtual="0x301" writable="1"/>
  <vif>
   <port real="8032" virtual="3632"/>
   <port real="8080" virtual="80"/>
  </vif>
 </vm>
 <instances>
  <jobinstance id="1" uid="1" host="amigos17.diku.dk"/>
  <jobinstance id="1" uid="2" host="amigos22.diku.dk"/>
 </instances>
</workload>
```

Figure 8.8: Example workload description in XML, for a VM with 256 megabytes of memory, and two ports open for incoming TCP connections.

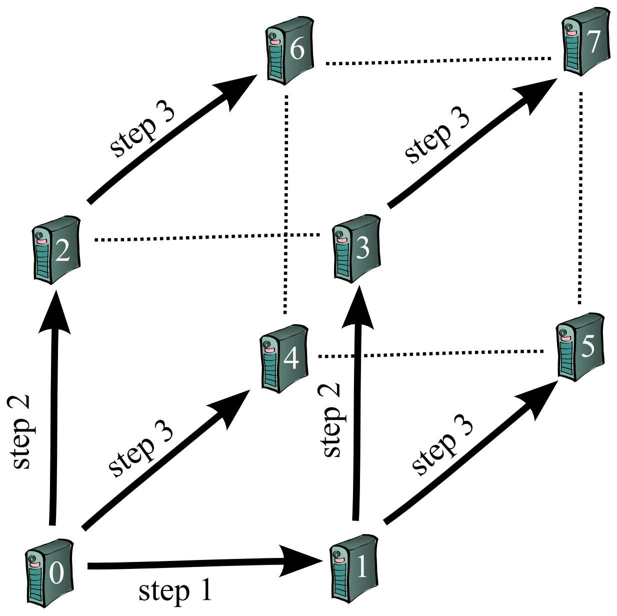


Figure 8.9: A three-dimensional hypercube with eight nodes. The root node is in the lower-left corner.

The first node (the root node) that boots is instantiated from a remote HTTP URI, whereas the remaining nodes are configured to listen for incoming connections. When the first node has completed booting, it connects to an empty VM on another node, and replicates itself there. This node in turn recursively replicates itself onto other nodes.

Each VM starts with a numerical identifier i as argument, *e.g.*, the root node has $i = 0$. As long there are un-booted nodes left, the destination i' is selected as $i' = i \vee 2^d$, where d is the currently active cube dimension. Figure 8.9 shows a cluster of eight nodes performing a three-dimensional hypercube broadcast.

Using a cluster of 16 machines, connected by a fully switched gigabit network, we measured the time taken to boot all nodes over a wide-area link. Each node was booted with a Linux kernel, a small initial ramdisk, and a 57MB root file system containing a Debian Linux distribution, compressed with SquashFS. The nodes were running at the University of Copenhagen, and the input data resided on a standard web server at the University of Aarhus, some 300 kilometers away. The bandwidth measured between the two institutions was approximately 9 Mb/s. The nodes in the cluster resided behind several layers of NAts and firewalls, and did not possess any routable IP addresses. The nodes were controlled using the "Evil Twin" model described in chapter 8.

In the star topology measurement all VMs booted over HTTP, and fetched their input data sets as a disk image, using the `wget` command line HTTP client. Once the disk image had been retrieved to a local disk partition, it was mounted, and the VM booted into an operational node, running `distcc` and a few other services. Because all VMs were booted at the same time, they contended for bandwidth on the wide-area network link, resulting in a long delay before the first VM became fully operational.

In the hypercube topology measurement, only a single VM booted using the `wget` method. Once the file system had been fetched, the VM initiated a hypercube broadcast of its kernel image, initial ramdisk, and the disk image. Every time a VM booted, it did the same, actively initiating connections to each node in its sub-cube. The disk images were read from and written to local disk, limiting the effective communication bandwidth to that of the local disks.

Figure 8.10 shows the number of fully booted VMs as a function of time, for the star and hypercube topologies. The time taken for the entire cluster to become operational is about three times shorter for the hypercube than for the star topology boot sequence. A larger input data set, a larger number of cluster nodes, or a wide-area link with less bandwidth, would make the difference even more pronounced. In addition, the bandwidth consumed on the wide-area link is 16 times higher for the star-topology boot than for the hypercube broadcast.

The intention of the graph is not to point out the obvious advantages of one form of network broadcast over another, but to show how the flexibility in configuring VMs yields real performance benefits.

Using similar mechanisms as above, we have used self-migration to perform a hypercube *fork*, to replicate a VM and its state to a large number of machines in a scalable way. Here, care must be taken to reconfigure globally visible identifiers, such as IP addresses. To show how this works in practice, we list the main loop of the hypercube-fork user-space driver in figure 8.11. As above, we tested the hypercube fork on our cluster of 16 identical cluster nodes. To monitor progress, we logged in to each of the nodes, using 16 X-terminals arranged in on the screen in a 4x4 grid. Figure 8.12 shows the 16 terminals before the fork. The node runs a Linux VM, while the remaining 15 nodes run only the `cstrap` in-VM firmware, listening for incoming TCP connections. Forking the 256MB VM across all nodes takes a little over one minute. Figure 8.13 shows the 16 VMs after the hypercube fork has completed. Except from the contents of the file `/tmp/myid`, the VMs are identical at this point.

Hypercube forks may be useful for applications with long warm-up phases, or guest OSes that take long to boot. One example that we have encountered is the POV-Ray ray-tracer[3], which spends several minutes building photon maps before rendering the actual image. The render process is trivial to parallelize, by rendering subparts of the image on each cluster node, but the warm-up phase is not. With a trivial modification of POV-Ray's main control function, we were able to insert a hypercube-fork between the two phases, saving the cost of the warm-up on all except the head node. The result was not returned faster, but the overall amount of CPU cycles spent to reach is was lower. For repeated invocations, the head node could process the warm-up phase of the next problem, while the slave nodes were busy with the rendering phase of the previous one.

8.5 Chapter Summary

In the chapter we have presented two different approaches to network management of large clusters. In both cases, our goal has been to minimize the amount of trusted software installed on each cluster node. Smaller implementations lead to systems that are more flexible and easier to secure, and thus likely to run longer without requiring operator attention.

Table 8.1 lists the main traits of our two models, and of the traditional model used in popular VMMs like Xen. In general, our models move functionality from the trusted VMM control plane, and into VMs. In both cases, our goal has been to minimize and simplify the trusted software installed on each cluster node. The Evil Man model reduces the privileged network facing code on each node to just a few hundred lines of C, providing a truly minimal network control plane implementation.

While the development of Xen ultimately forced us to include a full Linux kernel in our TCB, our new implementation tries to make the most of this situation. The "Evil Twin"

[3]http://www.povray.org

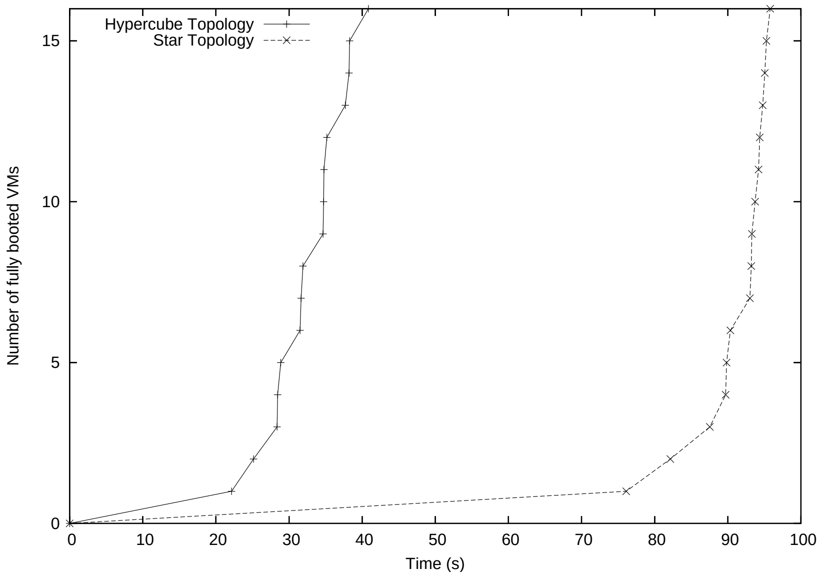
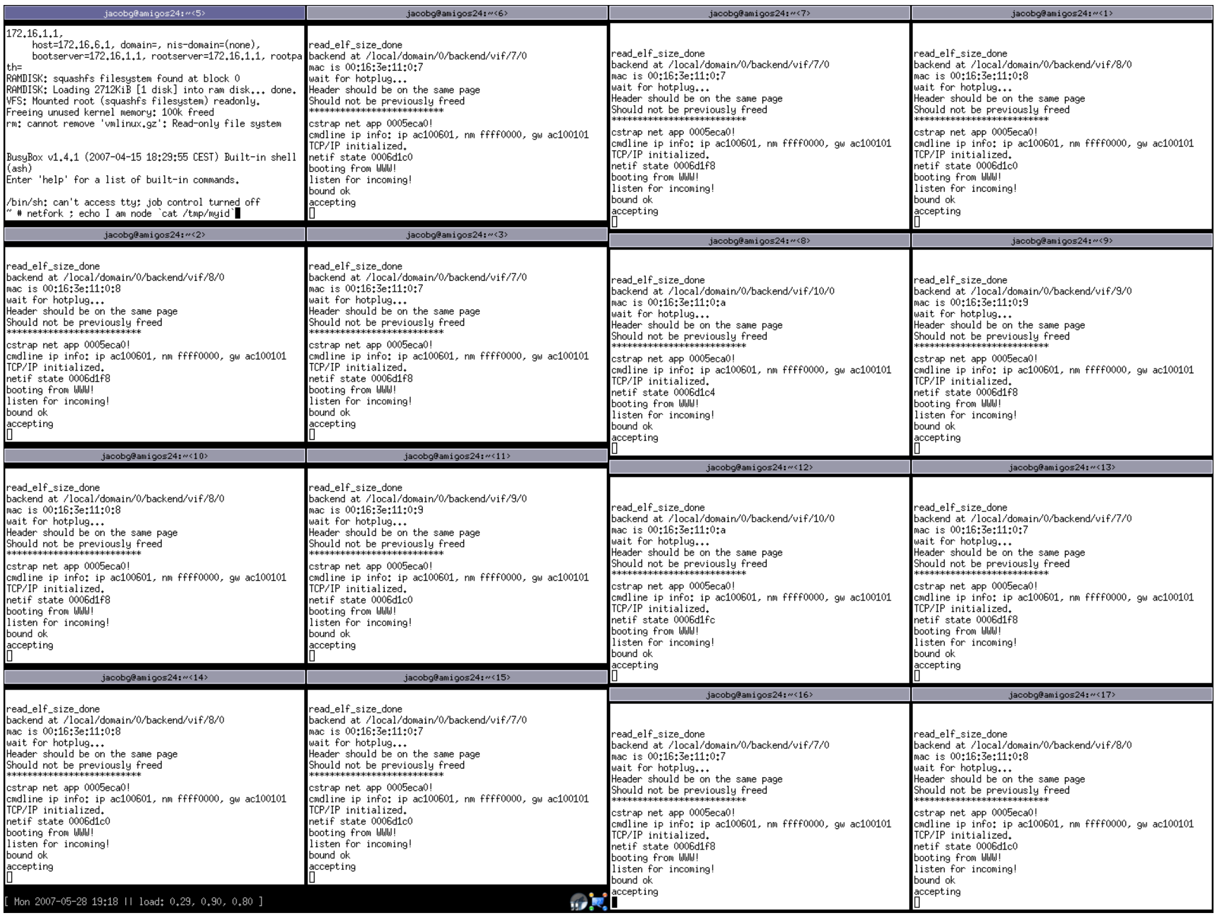


Figure 8.10: Number of VMs booted as a function of time, with either all VMs booting across the wide-area link (star topology), or with the first VM initiating a hypercube broadcast of VM images.

```
char* hosts[] = { "firsthost","secondhost",
                  "thirdhost","fourthhost", ... };
int dimension,my_id;

for(dimension=0,my_id=0; ; dimension++) {
  int sock, r;
  int dest_id = my_id | (1<<dimension);

  if(dest_id>=sizeof(hosts)/sizeof(char*)) exit(0);

  sock = connect_to_host(hosts[dest_id]);
  r = write_checkpoint(sock);
  close(sock);

  if( r==0 ) /* I am the child */ {
    my_id = dest_id;
    reconfigure_ip_address(my_id);
  }
}
```

Figure 8.11: Main loop of a VM-controlled hypercube fork. The body of the write_checkpoint function is similar to figure 6.3, and the return value tells the caller if it is the child or the parent of the network fork. If the VM is not subject to host-level NAT, it must reconfigure its IP address on the new host.



```
jacobg@amigos24:~<5>

172.16.1.1,
     host=172.16.6.1, domain=, nis-domain=(none),
     bootserver=172.16.1.1, rootserver=172.16.1.1, rootpa
th=
RAMDISK: squashfs filesystem found at block 0
RAMDISK: Loading 2712KiB [1 disk] into ram disk... done.
VFS: Mounted root (squashfs filesystem) readonly.
Freeing unused kernel memory: 100k freed
rm: cannot remove 'vmlinux.gz': Read-only file system


BusyBox v1.4.1 (2007-04-15 18:29:55 CEST) Built-in shell
(ash)
Enter 'help' for a list of built-in commands.

/bin/sh: can't access tty: job control turned off
~ # netfork ; echo I am node `cat /tmp/myid`
```

```
jacobg@amigos24:~<6>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<7>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<1>

read_elf_size_done
backend at /local/domain/0/backend/vif/8/0
mac is 00:16:3e:11:0:8
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<2>

read_elf_size_done
backend at /local/domain/0/backend/vif/8/0
mac is 00:16:3e:11:0:8
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<3>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<8>

read_elf_size_done
backend at /local/domain/0/backend/vif/10/0
mac is 00:16:3e:11:0:a
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c4
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<9>

read_elf_size_done
backend at /local/domain/0/backend/vif/9/0
mac is 00:16:3e:11:0:9
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<10>

read_elf_size_done
backend at /local/domain/0/backend/vif/8/0
mac is 00:16:3e:11:0:8
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<11>

read_elf_size_done
backend at /local/domain/0/backend/vif/9/0
mac is 00:16:3e:11:0:9
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<12>

read_elf_size_done
backend at /local/domain/0/backend/vif/10/0
mac is 00:16:3e:11:0:a
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1fc
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<13>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<14>

read_elf_size_done
backend at /local/domain/0/backend/vif/8/0
mac is 00:16:3e:11:0:8
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

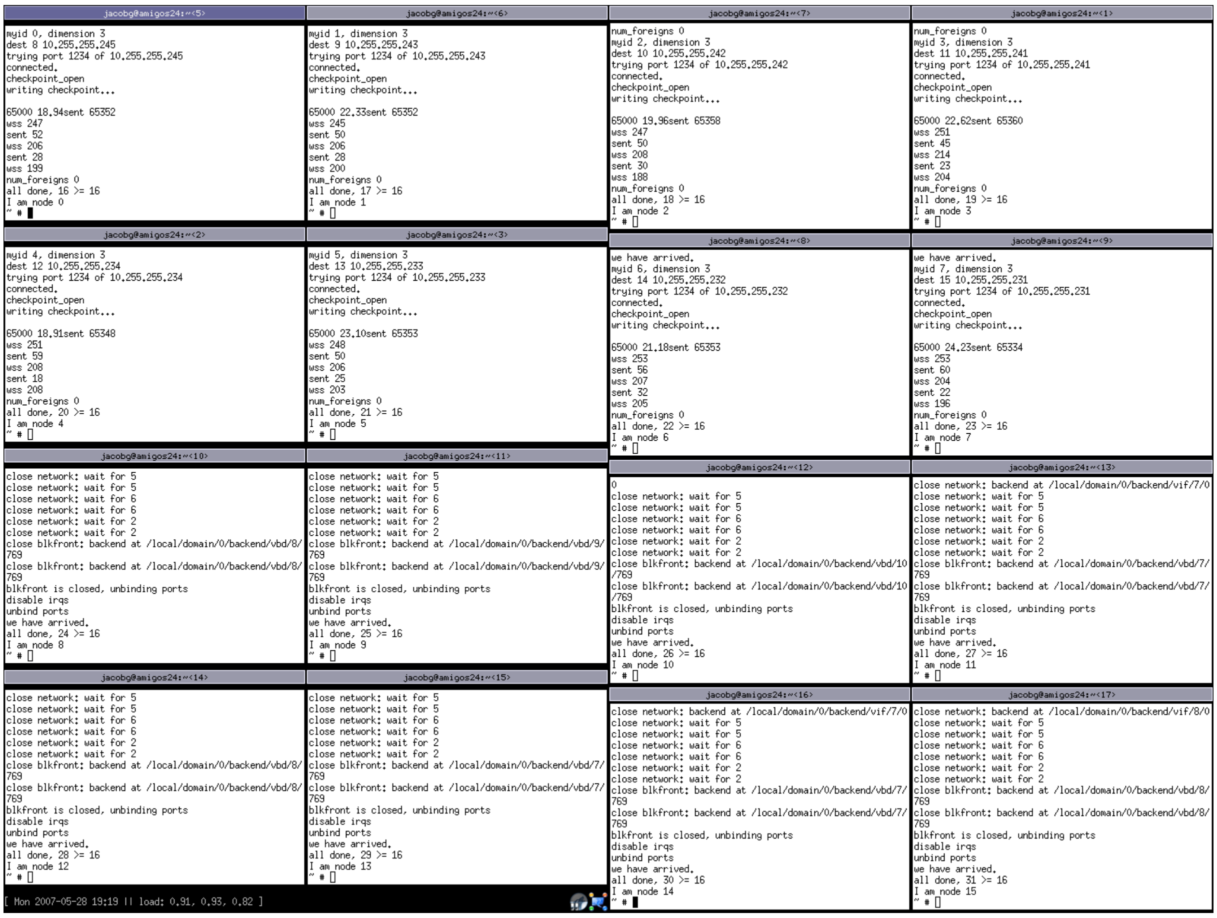
```
jacobg@amigos24:~<15>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<16>

read_elf_size_done
backend at /local/domain/0/backend/vif/7/0
mac is 00:16:3e:11:0:7
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1f8
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
jacobg@amigos24:~<17>

read_elf_size_done
backend at /local/domain/0/backend/vif/8/0
mac is 00:16:3e:11:0:8
wait for hotplug...
Header should be on the same page
Should not be previously freed
**************************
cstrap net app 0005eca0!
cmdline ip info: ip ac100601, nm ffff0000, gw ac100101
TCP/IP initialized.
netif state 0006d1c0
booting from WWW!
listen for incoming!
bound ok
accepting
```

```
[ Mon 2007-05-28 19:18 || load: 0.29, 0.90, 0.80 ]
```

Figure 8.12: Before a hypercube fork across 16 cluster nodes. The upper-left window shows the single active VM, before initiating a hypercube fork.



```
jacobg@amigos24:~<5>
myid 0, dimension 3
dest 8 10.255.255.245
trying port 1234 of 10.255.255.245
connected.
checkpoint_open
writing checkpoint...

65000 18.94sent 65352
wss 247
sent 52
wss 206
sent 28
wss 199
num_foreigns 0
all done, 16 >= 16
I am node 0
~ #
```

```
jacobg@amigos24:~<6>
myid 1, dimension 3
dest 9 10.255.255.243
trying port 1234 of 10.255.255.243
connected.
checkpoint_open
writing checkpoint...

65000 22.33sent 65352
wss 245
sent 50
wss 206
sent 28
wss 200
num_foreigns 0
all done, 17 >= 16
I am node 1
~ #
```

```
jacobg@amigos24:~<7>
num_foreigns 0
myid 2, dimension 3
dest 10 10.255.255.242
trying port 1234 of 10.255.255.242
connected.
checkpoint_open
writing checkpoint...

65000 19.96sent 65358
wss 247
sent 50
wss 208
sent 30
wss 188
num_foreigns 0
all done, 18 >= 16
I am node 2
~ #
```

```
jacobg@amigos24:~<1>
num_foreigns 0
myid 3, dimension 3
dest 11 10.255.255.241
trying port 1234 of 10.255.255.241
connected.
checkpoint_open
writing checkpoint...

65000 22.62sent 65360
wss 251
sent 45
wss 214
sent 23
wss 204
num_foreigns 0
all done, 19 >= 16
I am node 3
~ #
```

```
jacobg@amigos24:~<2>
myid 4, dimension 3
dest 12 10.255.255.234
trying port 1234 of 10.255.255.234
connected.
checkpoint_open
writing checkpoint...

65000 18.91sent 65348
wss 251
sent 59
wss 208
sent 18
wss 208
num_foreigns 0
all done, 20 >= 16
I am node 4
~ #
```

```
jacobg@amigos24:~<3>
myid 5, dimension 3
dest 13 10.255.255.233
trying port 1234 of 10.255.255.233
connected.
checkpoint_open
writing checkpoint...

65000 23.10sent 65353
wss 248
sent 50
wss 206
sent 25
wss 203
num_foreigns 0
all done, 21 >= 16
I am node 5
~ #
```

```
jacobg@amigos24:~<8>
we have arrived.
myid 6, dimension 3
dest 14 10.255.255.232
trying port 1234 of 10.255.255.232
connected.
checkpoint_open
writing checkpoint...

65000 21.18sent 65353
wss 253
sent 56
wss 207
sent 32
wss 205
num_foreigns 0
all done, 22 >= 16
I am node 6
~ #
```

```
jacobg@amigos24:~<9>
we have arrived.
myid 7, dimension 3
dest 15 10.255.255.231
trying port 1234 of 10.255.255.231
connected.
checkpoint_open
writing checkpoint...

65000 24.23sent 65334
wss 253
sent 60
wss 204
sent 22
wss 196
num_foreigns 0
all done, 23 >= 16
I am node 7
~ #
```

```
jacobg@amigos24:~<10>
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 24 >= 16
I am node 8
~ #
```

```
jacobg@amigos24:~<11>
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/9/
769
close blkfront: backend at /local/domain/0/backend/vbd/9/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 25 >= 16
I am node 9
~ #
```

```
jacobg@amigos24:~<12>
0
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/10
/769
close blkfront: backend at /local/domain/0/backend/vbd/10
/769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 26 >= 16
I am node 10
~ #
```

```
jacobg@amigos24:~<13>
close network: backend at /local/domain/0/backend/vif/7/0
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 27 >= 16
I am node 11
~ #
```

```
jacobg@amigos24:~<14>
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 28 >= 16
I am node 12
~ #
```

```
jacobg@amigos24:~<15>
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 29 >= 16
I am node 13
~ #
```

```
jacobg@amigos24:~<16>
close network: backend at /local/domain/0/backend/vif/7/0
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
close blkfront: backend at /local/domain/0/backend/vbd/7/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 30 >= 16
I am node 14
~ #
```

```
jacobg@amigos24:~<17>
close network: backend at /local/domain/0/backend/vif/8/0
close network: wait for 5
close network: wait for 5
close network: wait for 6
close network: wait for 6
close network: wait for 2
close network: wait for 2
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
close blkfront: backend at /local/domain/0/backend/vbd/8/
769
blkfront is closed, unbinding ports
disable irqs
unbind ports
we have arrived.
all done, 31 >= 16
I am node 15
~ #
```

```
[ Mon 2007-05-28 19:19 || load: 0.91, 0.93, 0.82 ]
```

Figure 8.13: One minute later. The 256MB VM has forked and now runs on all 16 nodes. Each node prints its id after arrival.

Model	**Evil Man**	**Evil Twin**	**Traditional**
Control Model	Decentralized	Centralized	Centralized
Accounting	Tokens	None	None
Implementation Complexity			
Code Size (LOC)	500	Thousands	Millions
Controller Implementation	Complex	Simple	Complex
Networking			
VM Self-Replication	Yes	Yes	No
Control nodes behind NATs	No	Yes	No
Bootstrap			
Guest Image Decoding	In VM	In VM	In TCB
Migration			
Live Migration	Yes	Yes	Yes
Transparent Migration	No	No	Yes
Checkpoint Send	In VM	In VM	In TCB
Checkpoint Receive	In VM	In VM	In TCB
Encryption			
Type of Encryption	Shared Secret	SSL	SSL

Table 8.1: The three network control plane models compared.

control plane never accepts incoming connections, and the use of unforgeable Pulse events means that a node cannot be tricked into initiating connections. We remain hopeful that future hypervisor developments will allow us to remove the Linux kernel. If this becomes the case, we can move the other complex parts of Evil Twin into an unprivileged VM, talking to Evil Man's `packetd` inside the privileged VM using the ICMP boot token protocol.

When multiple VMs are on the same local network, it is possible to speed up the deployment of input data sets through some form of network broadcast. We have shown how the use of VMs makes this relatively simple, by letting the first VM that enters the network perform a hypercube broadcast of its state to all nodes on the network, and how this reduces the startup time for a parallel task involving 16 nodes.

CHAPTER 9

Virtual Machines on the Desktop

The motivation for the work described in this chapter is a desire to explore the potential uses of VMs and self-checkpointing in a desktop setting. The benefits of VMs in this case would be stronger isolation, and the ability to checkpoint individual applications to stable storage. Today, a user may be able to checkpoint the state of his laptop onto the local disk, and to resume from this state at a later point. Our vision is for a system similar to a bookmark database, but with a complete application checkpoint per entry instead of a simple URL text string. In order for such a system to be able to host modern applications that require high graphical throughput, and still be able to checkpoint these, a new display system is required. Currently deployed display systems for VMs either do not support 3D acceleration, or are unable to checkpoint state downloaded to the graphics adapter. Other necessary components, that we have yet to implement, include efficient delta compression for checkpoints (similar to the system described by Nath et al. [2006]), and a mechanism for the exchange of data objects between application VMs.

9.1 The Blink Display System

In addition to their popularity in data centers, Virtual Machines (VMs) are increasingly deployed on client machines, *e.g.*, to allow for compartmentalization of untrusted software downloaded from the Internet [Cox et al., 2006], or for ease of management [Chandra et al., 2005]. VM technology is quickly becoming commoditized, and it is conceivable that future desktop operating systems will ship with VM-style containers as a standard feature.

While initially users may be willing to put up with seeing multiple desktops in a simple picture-in-picture fashion, over time the demand for more integrated and seamless experiences will grow. Users will expect virtualized applications to blend in nicely, and will want to make use of graphics hardware acceleration, for games, simulations, and video conferencing. Our work attempts to address this need by providing virtual machines with access to the powerful accelerated drawing features of modern display hardware, without compromising the safety guarantees of the VM model.

Compared to other I/O subsystems, the display system is harder to multiplex in a way that is both efficient and safe, especially for demanding applications such as 3D games or full-screen video. This is evidenced by the fact that the major operating systems all

Figure 9.1: Blink running GLGears and MPlayer.

provide “direct” avenues of programming the graphics card, largely without operating system involvement, but at the danger of being able to crash the graphics adapter or lock up the entire machine [Faith and Martin, 1999]. Because of this danger, untrusted software running inside VMs should not be given direct hardware access, and a layer of indirection between client and hardware is necessary. Such a layer should also provide a hardware-independent abstraction, to allow a VM to run unmodified across different types of graphics adapters. One way of implementing this layer is by letting clients program to a high-level API, and have a trusted display system translate API commands into programming of the actual hardware. The trusted translation step verifies the safety of each API command, and then forwards commands to the graphics card driver. In this way, applications can be prevented from bypassing address space protection with malicious DMA operations, or exploiting bugs in graphics hardware or driver APIs. All use of hardware resources can be tracked and subjected to a policy that prevents one application from causing starvation, *e.g.*, by consuming all memory on the graphics card.

This chapter describes Blink, a prototype system for providing untrusted VMs with safe and efficient access to modern display hardware. Blink aims to be as safe, simple and flexible as possible, while at the same time retaining full performance.

The goal of Blink is to serve as the display component of a system where each application is encapsulated inside its own VM. Blink is backwards-compatible with existing software, through emulation layers for X11, kernel framebuffers, or OpenGL, but performance may be enhanced by adjusting applications to bypass these layers and use Blink directly.

The Blink prototype runs on top of the Xen [Barham et al., 2003] virtual machine monitor. The Blink display server runs as a regular user-application inside a Linux guest

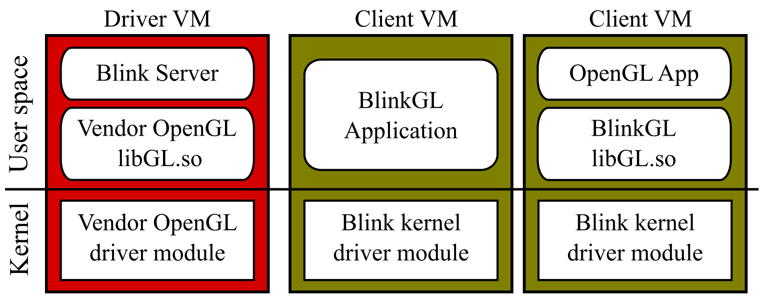


Figure 9.2: Blink client VMs communicate with the server VM using shared memory. Dedicated BlinkGL clients access the BlinkGL primitives directly, and unmodified OpenGL clients go through an OpenGL driver library.

VM, using commercially developed device drivers for graphics hardware access. The VM running the Blink server mediates access to the graphics hardware, and clients running inside untrusted VMs talk to this server using a shared-memory protocol, as shown in figure 9.2.

In the remainder of this section, we first briefly introduce 3D programming using OpenGL, and the challenges faced when trying to make an OpenGL-like abstraction to clients in separate protection domains. From there, we describe the key points of our design, and how they have been implemented in our prototype.

9.1.1 GPU Programming

Most modern GPU programming is done using APIs such as OpenGL [Kilgard, 1997] or Direct3D [Blythe, 2006]. Our work focuses on OpenGL, a standardized API that is available on most platforms. An OpenGL (GL) program is a sequence of API-calls, of the form *glName(args)*, where *Name* is the name of the called GL command. Some commands modify state such as transformation matrices, lighting and texturing parameters, and some result in immediate drawing to the screen. Drawing commands are enclosed in *glBegin()* and *glEnd()* pairs, *e.g.*, the following GL program draws a triangle:

```
glBegin(GL_TRIANGLES);
glVertex(0,0);
glVertex(1,0);
glVertex(1,1);
glEnd();
```

An OpenGL API implementation is often split into two parts. The kernel part provides primitives for accessing graphics card DMA buffers and hardware registers, and the user space part—a GPU-specific shared library known as the Installable Client Driver (ICD)—accesses these primitives to translate GL API calls into programming of the

GPU. Thus, the output of an OpenGL program can be re-targeted to new hardware, or to a virtualized abstraction, by replacing the ICD installation files.

9.1.2 Off-Screen Buffers

In a traditional windowing system, each application owns one or more clearly defined rectangular regions of the screen, and such systems expend much effort clipping drawing operations to correctly fall within these regions. With the advent of faster graphics cards with greater amounts of video memory, a new model has become dominant. Pioneered by the MacOSX "Quartz" composited display system, the new approach is to give each application its own *off-screen buffer*, a contiguous range of video memory to which the application may draw freely. The display system runs a special *compositor* process which takes care of painting all of the off-screen buffers to the screen, in back-to-front order. The benefits of this model are simplicity and the support for advanced effects such as translucent or zooming windows. The problem with this model is the great amount of memory required for off-screen buffers, to avoid context-switching back and forth between the server and all clients every time the screen is redrawn. There are also cases where off-screen buffers are likely to contain redundant information, *e.g.*, in the case of a video player. A video player will often decode the current video frame to a texture with the CPU, then use the GPU to display the texture mapped onto a rectangular surface. If the video player is forced to go via an off-screen buffer, the frame will have to be copied twice by the GPU, first when it is drawn to the off-screen buffer, and second when the display is composed by the display system. In our proposed system, the decision of whether and how to use off-screen buffers is left to applications. As we shall see, this flexibility is achieved by letting applications execute code as *Stored Procedures (SPs)* inside the display server.

9.1.3 BlinkGL Stored Procedures

Virtual machine systems often exploit existing wire protocols when communicating between a VM and the surrounding host environment. For instance, a remote desktop protocol such as RDP or VNC makes connecting a VM to the display relatively straightforward. VNC and RDP communicate screen updates in terms of pixels and rectangles. Remote display of 3D may be achieved in these systems by rendering the final image server-side [De Winter et al., 2006], though at an additional cost in bandwidth, and with the risk of server CPU or GPU contention.

When wishing to render 3D content locally on the client, the common solution is to serialize rendering commands, into batches of Remote Procedure Calls (RPCs [Birrell and Nelson, 1984]), which are then communicated over the wire protocol. In addition to the cost of communication, this method carries a performance overhead, due to the cost of serializing and de-serializing command streams. In OpenGL, translation costs can be amortized to some extent by the use of *display lists*, macro-sequences of GL commands stored in video memory. However, display lists are static and only useful in the parts of the GL program that need not adapt to frequently changing conditions. Blink extends the display list abstraction into more general and flexible *BlinkGL* stored procedures. Because stored procedures are richer in functionality than display lists, they can handle simple user interactions, *e.g.*, redrawing the mouse cursor or highlighting a pushbutton in response to a mouse rollover—independently of the application VM.

BlinkGL is a super-set of OpenGL. BlinkGL stored procedures run on the CPU—inside the display server—and in addition to GL commands they can also perform simple arithmetic and control operations. Stored procedures are sequences of serialized BlinkGL commands, with each command consisting of an opcode and a set of parameters. A part of the opcode space is reserved for special operations for virtual register copying, arithmetic, or conditional forward-jumps. External state, such as mouse coordinates or window dimensions, can be read into registers with special BlinkGL calls, processed, and the results given as arguments to other BlinkGL calls that take register arguments. The Blink server contains a Just-In-Time (JIT) compiler that converts BlinkGL into native CPU machine code that is invoked during screen redraw or in response to user input. Because of the simplicity of BlinkGL, JIT compilation is fast, and for GL calls the generated code is of similar quality to the output of a C-compiler. Apart from amortizing translation costs, the use of SPs also has two additional benefits: CPU context switching is greatly reduced because each client does not have to be consulted upon every display update, and in many cases the use of off-screen buffers to hold client areas can be avoided by drawing client areas with SPs on the fly.

Table 9.1 lists the most common SPs that a Blink client will provide.

9.1.4 Program Inspection

Some aspects of SP execution require additional checking, *e.g.*, to prevent clients drawing outside of their windows. During JIT compilation, commands and parameters are inspected, and filtered or adjusted according to various policies. This inspection allows the Blink server to weed out unwanted client actions, but also enables global optimizations that exploit advance knowledge of client behavior. If the client is known not to use the Z-buffer for depth-ordering of drawing operations, then the Z-buffer does not need to be cleared before invoking the client's redraw code, and if the client is not enabling transparency, content covered by the client's rectangle does not need to be redrawn. The constraints placed on a SP during compilation depend on how the SP is used, as not all callbacks are allowed to perform operations that draw to or clear the screen. During compilation, the safety of each SP command is checked, so that out-of-bounds parameters or illegal command sequences may be detected before the SP is allowed to run. Out-of-bounds parameters are modified to be safe, and unsafe commands are skipped altogether. This ensures that once compiled, SPs are safe to execute within the context of the Blink server.

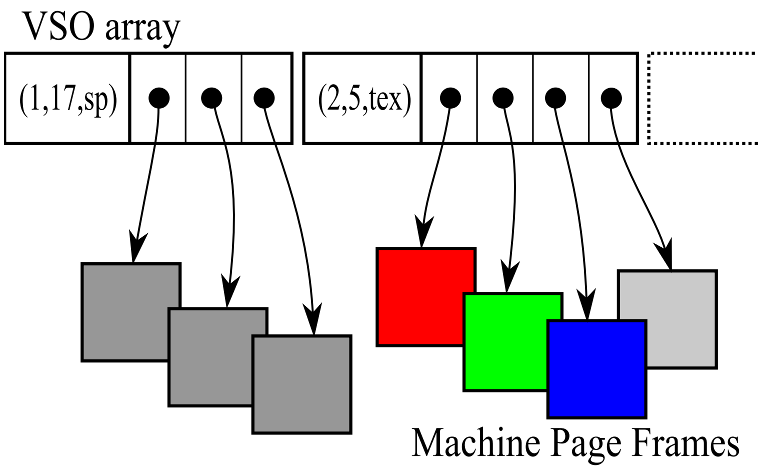


Figure 9.3: Versioned Shared Objects with OID's 1 and 2 respectively, pointing to physical machine page frames. The first object contains a BlinkGL stored procedure, and the second a texture image.

This static analysis is simple and conservative; it does not attempt to predict the outcomes of conditionals. Instead it errs on the side of safety and only assumes that a given mandatory command will execute if it is outside of any conditional scope. The reverse is true for commands that may have dangerous side-effects—they are assumed to always execute regardless of any enclosing conditional scopes.

9.1.5 Versioned Shared Objects

VMs residing on the same physical host may communicate through shared memory, instead of using wire protocols that are likely to introduce extra data copying, especially problematic for large texture or framebuffer objects. Client VMs communicate with Blink through an array of Versioned Shared Objects (VSO's). A VSO is an in-memory data record containing an object identifier (OID), an object type identifier, a version number, and a list of memory pages containing object data. When Blink receives an update notification from a client VM, it scans through the client's VSO array, looking for updated objects. When a changed or new object is encountered, Blink performs type-specific processing of object contents, such as JIT compilation of stored procedures, and incorporates any changes in the next display update. Each VM may maintain several VSO arrays, to accommodate the use of multiple OpenGL hardware contexts for different clients within the VM, and care is taken to avoid scanning unmodified arrays. The scan is linear in the number of VSO's in the array, but more elaborate data structures can be envisioned if the scalability of the current approach becomes an issue. Figure 9.3 shows the first two objects in a VSO array containing a stored procedure and a texture object. Most GL commands are called directly by the JIT'ed machine code, but commands taking pointer arguments are treated specially. For example, the *glTexImage2D()* command uploads a texture from a main memory address to the graphics card. Blink's version of the command instead takes a VSO OID, which is then resolved into a memory address during compilation. Texture data gets DMA'ed from application to video memory, and inter-VM copying is avoided.

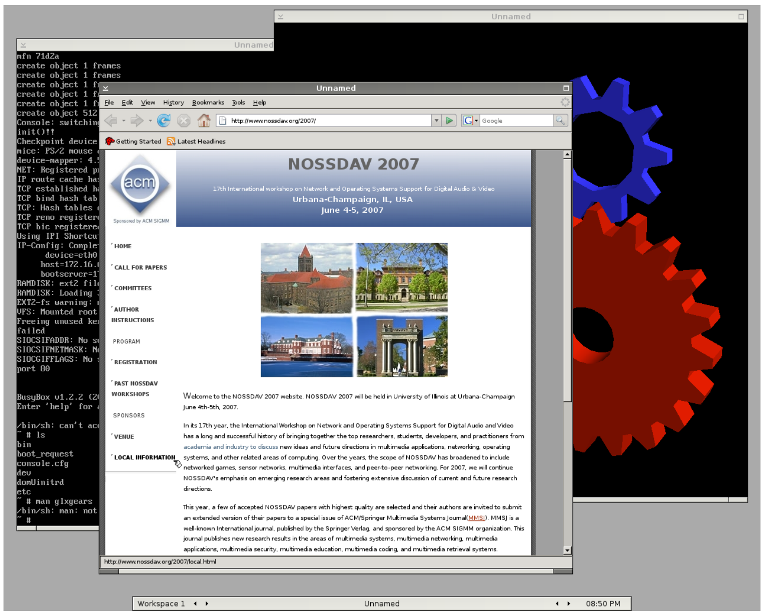
9.1.6 Virtualizing Standard OpenGL

Rather than expecting all existing OpenGL software to be rewritten as BlinkGL, Blink also contains a compatibility wrapper which allows unmodified OpenGL software to display via Blink. This wrapper is implemented as a custom, client-side ICD, with the help of an additional BlinkGL command, called *glEval()*, in the Blink server.

The *glEval* command invokes an interpreter that understands serialized standard OpenGL and executes it immediately, and by combining client-side driver code with a server-side SP in a producer-consumer pair, it is possible to transparently host unmodified OpenGL software. Like other display systems that compose multiple OpenGL clients to a shared display, the wrapper needs off-screen buffers to avoid flicker or artifacts, and to allow arbitrary effects such as transparent windows. However, this functionality is not supported by the Blink display server. Instead, SPs that are part of the client ICD handle this by executing *glEval()* in the context of an off-screen buffer, and drawing the off-screen buffer onto a texture during the redraw SP. This way, Blink is able to host unmodified OpenGL applications, and to subject them to arbitrary transformations when composing the screen.

For the code interpreted by *glEval()*, the overhead of full JIT compilation is not justifiable. Instead, Blink implements a specialized interpreter for serialized OpenGL streams. When writing an interpreter, the choice is basically between implementing a "giant switch" with a case for each opcode number, or the "threaded code" approach, with a jump-table with an entry for each opcode number, both of which may be implemented as portable C code. On modern architectures, both approaches suffer from poor branch prediction accuracy, as low as 2%-50%, due to a large number of indirect branches [Ertl and Gregg, 2003]. Furthermore, the ratio of arguments to opcodes is high for serialized OpenGL, so a traditional interpreter has to spend much of its time copying arguments from the input program to the parameter stack of the called GL API function. As a more efficient alternative, we designed a new interpretation technique which we refer to as *Stack Replay*.

Stack Replay is a simple and fast interpretation technique designed specifically for the characteristics of OpenGL and similar sequences of batched remote procedure calls. It is based on the observation that a serialized OpenGL program is little more than an array of call-stacks. This means that parameter copying can be avoided altogether by pointing the stack pointer directly to opcode parameters before each API call. Branch prediction can be improved by using a minimal code-generator which converts input programs into machine code sequences of API calls interleaved with increments of the stack pointer register, so that arguments are consumed directly from the input pro-



```
mfn 71d2a
create object 1 frames
create object 1 frames
create object 1 f
create object 1 f
create object 1 f
create object 512
Console: switchin
init()!!
Checkpoint device
mice: PS/2 mouse
device-mapper: 4.
NET: Registered p
IP route cache ha
TCP established h
TCP bind hash tab
TCP: Hash tables
TCP reno register
TCP bic registere
Using IPI Shortcu
IP-Config: Comple
      device=eth0
     host=172.16.
     bootserver=1
RAMDISK: ext2 fil
RAMDISK: Loading
EXT2-fs warning:
VFS: Mounted root
Freeing unused ke
failed
SIOCSIFADDR: No s
SIOCSIFNETMASK: N
SIOCGIFFLAGS: No
port 80


BusyBox v1.2.2 (2
Enter 'help' for

/bin/sh: can't ac
~ # ls
bin
boot_request
console.cfg
dev
domUinitrd
etc
~ # man glxgears
/bin/sh: man: not
~ #
```

NOSSDAV 2007

17th International workshop on Network and Operating Systems Support for Digital Audio & Video
Urbana-Champaign, IL, USA
June 4-5, 2007

Welcome to the NOSSDAV 2007 website. NOSSDAV 2007 will be held in University of Illinois at Urbana-Champaign June 4th-5th, 2007.

In its 17th year, the International Workshop on Network and Operating Systems Support for Digital Audio and Video has a long and successful history of bringing together the top researchers, students, developers, and practitioners from academia and industry to discuss new ideas and future directions in multimedia applications, networking, operating systems, and other related areas of computing. Over the years, the scope of NOSSDAV has broadened to include networked games, sensor networks, multimedia interfaces, and peer-to-peer networking. For 2007, we will continue NOSSDAV's emphasis on emerging research areas and fostering extensive discussion of current and future research directions.

This year, a few of accepted NOSSDAV papers with highest quality are selected and their authors are invited to submit an extended version of their papers to a special issue of ACM/Springer Multimedia Systems Journal(MMSJ). MMSJ is a well-known International journal, published by the Springer Verlag, and sponsored by the ACM SIGMM organization. This journal publishes new research results in the areas of multimedia systems, multimedia networking, multimedia applications, multimedia security, multimedia education, multimedia coding, and multimedia retrieval systems.

Figure 9.4: Blink running in root-less mode on top of X11. Here, the kernel framebuffer is shown, both running a text mode console and with X11 in a VM, running a web browser.

gram without copying.[1] This approach is platform-specific, but does offer better performance than platform neutral alternatives such as a giant-switch interpreter, which could still be provided as a fall-back on other platforms. The platform-specific parts of the interpreter consist of a six line main-loop in C, and 10 lines of inline x86 assembly pre- and postambles, used when calling a batch of generated code.

When using Linux as the guest VM, the Blink client driver also supports displaying the Linux kernel framebuffer and X11 on an OpenGL texture. This is accomplished by adding a framebuffer driver, which maps a main memory buffer onto a BlinkGL texture, to the guest Linux kernel. Using this driver it is possible to run legacy text mode and X11 applications on top of Blink. Figure 9.4 shows X11 running on top of a BlinkGL texture.

9.1.7 Frame Rate Independence

Often, application content must be adjusted to fit the system screen refresh rate. For instance, a sequence of video frames encoded at 50fps can be adjusted to a 70fps display by repeating some frames two times, but this uneven display rate will make the video

[1]The reader will notice that using the stack in this manner destroys the input program, and that space must be available at the head of the program, as otherwise the stack will overflow. The first is not a problem, because programs need only be interpreted once. The latter we deal with by proper use of virtual memory mappings.

Callback Name	Executed
`init()`	At first display
`update()`	On client VM request
`reshape()`	On window move or resize
`redraw()`	For each display
`input()`	On user input

Table 9.1: List of client Stored Procedures and their times of invocation.

appear choppy. An alternative is to interpolate content, for instance by blending subsequent video frames into artificial intermediate frames. Blink SPs can perform simple interpolation and blending, by rendering multiple frames over another with variable transparency. Another advantage of Blink SPs here is that multiple frames may be batch-decoded during the VMs time slice, and then later rendered to the display by the SP. This will lead to fewer context switches and improved system throughput. For 3D content, such *frame rate independence* may be achieved simply by specifying 3D object positions as functions of time instead of as constants hard-coded in GL command arguments. Simple animations such as the classic spinning gears in the GLXGears example application may run mostly independent of the client VM, with gear angles specified as functions of time.

9.1.8 Display State Recovery

The precursor to Blink was the 2D Tahoma [Cox et al., 2006] Display System (TDS). TDS was completely stateless, with the display acting merely as a cache of client content. Among other benefits, this allowed for checkpointing and migration [Clark et al., 2005] of VMs. BlinkGL clients can be implemented to be stateless, *e.g.*, the server just calls the client's initialization SP to recreate any missing state, but the transparently virtualized OpenGL programs are not stateless out of the box, because OpenGL is itself a stateful protocol. To solve this problem, we have added a state-tracking facility to the client-side OpenGL emulation layer, so that copies of all relevant state are maintained inside the address space of the calling application. Display lists and textures are captured verbatim, and the effects of transformation to the OpenGL matrix stack are tracked, so that they can later be replayed. Our approach here is in many ways similar to the one described by Buck et.al. [Buck et al., 2000], but with the purpose of being able to recreate lost state, rather than as a performance optimization.

9.2 Evaluation

In this section we attempt to measure key aspects of Blink's performance. At the micro-level we measure the overheads introduced by JIT compilation and interpretation, and at the macro-level we measure overall system throughput for a simple application. These benchmarks do not claim to be an exhaustive evaluation, but they give a good indication of how Blink performs relative to native OpenGL code.

We evaluated Blink on a low-end desktop PC, a Dell Optiplex GX260 PC with a 2GHz single-threaded Intel Pentium4 CPU, with 512kB cache and 1024MB SDRAM. The machine was equipped with an ATI Radeon 9600SE 4xAGP card with 128MB DDR RAM, using ATI's proprietary OpenGL display driver.

We first evaluated JIT compiler performance. We instrumented the compiler to read the CPU time stamp counter before and after compilation, and report average number of CPU cycles spent per input BlinkGL instruction. Because we did not measure OpenGL performance in this test, all call-instructions emitted by the compiler point to a dummy function which simply returns. We measured instructions spent per executed virtual instruction, and report per virtual instruction averages.

As input we created two programs; the first (OpenGL-mix) is the setup phase of the GLGears application, repeated six times. This program performs various setup operations, followed by upload of a large amount of vertexes for the gear objects with the *glVertex()* command. The second (Arith-mix) is mix of 8 arithmetic operations over two virtual registers, repeated for every combination of the 32 virtual registers. Both programs consist of roughly 8K instructions, performance figures are in table 9.2. We noticed that subsequent invocations (numbers in parentheses) of the compiler were almost twice as fast as the first one, most likely because of the warmer caches on the second run. We expect larger programs and multiple SPs compiled in sequence to see similar performance gains. Arithmetic operations on virtual floating point registers are costlier than GL calls, as we make little attempt to optimize them. Finally, we measure the cost of interpretation. We call the interpreter with an input program similar to OpenGL-mix, and measure the cost of interpretation using Stack Replay. As before, all calls are to dummy functions that simply return immediately. We see that the use of interpretation adds about 44% overhead compared with the execution of JIT compiled code (59 vs. 41 cpi respectively). Thus there is benefit to using the JIT compiler in cases where the cost can be amortized, but the interpreter yields reasonable performance as well.

To validate our claim that for GL-calls the JIT compiler produces code of similar-quality as `gcc`, we also ran the OpenGL-mix code in two scenarios—statically compiled into the display server, and in the JIT compiled version. Performance of the two programs is nearly identical, as can be seen in table 9.3.

Secondly we measured overall system throughput. For this we ported two applications, the classic GLGears demo, and the popular open-source video player "MPlayer", to display using Blink Stored Procedures. GLGears displays three spinning gears 3D rendered into a 512x512 window. We ran multiple GLGears instances, each in a separate VM, and measured the average time deltas between client updates.

MPlayer decodes video frames from a 512x304 resolution DivX file to a memory buffer,

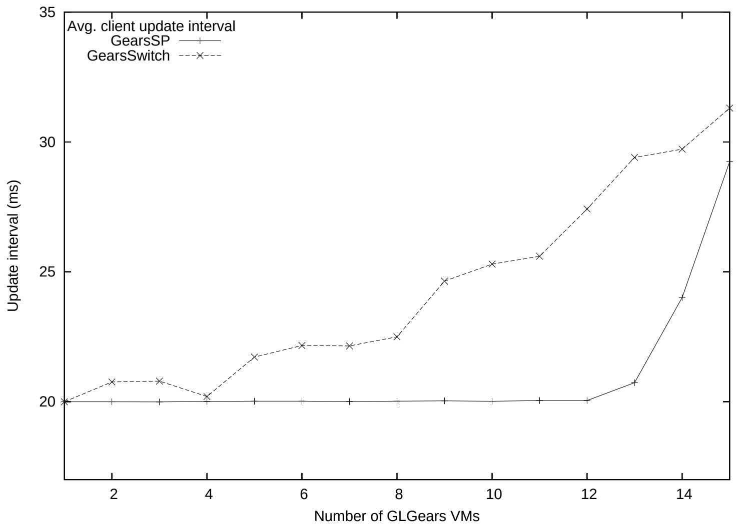


Figure 9.5: Averaged delay between frame updates of the OpenGL Gears demo. GearsSP uses stored procedures; GearsSwitch context switches between VMs.

Type of input	#Instr.	Compile	Execute
OpenGL JIT	8,034	102 (41) cpi	41 cpi
Arithmetic JIT	8,192	99 (55) cpi	50 cpi
OpenGL interpr.	8,191	0 cpi	59 cpi

Table 9.2: Cycles-per-instruction for the JIT compiled SPs, and for interpreted OpenGL streams, on a 2GHz Pentium 4. Numbers in parentheses with warm cache.

which is copied to a texture by the `update` SP, and composed to the screen by the `redraw` SP. We measured time deltas in both SPs, because the former is only run on request by the client, whereas the latter runs every time the shared screen is redrawn. Times for `update` are a measure of the system level of service as seen by each client, where times for `redraw` are a measure of display server responsiveness. MPlayer shows a video running at 25fps, so `update` and `redraw` should be serviced at 40ms intervals. Figure 9.6 shows time deltas for `update` and `redraw` as functions of the number of client VMs executing. For up to 5 concurrent MPlayer instances, the system is able to provide adequate service to client VMs. However, for up to 12 VMs (at which point the machine ran out of memory), the `redraw` SP is still serviced at more than the required rate. We expect this to be partly due to scheduler interaction between MPlayer's timing code, Blink, and Xen, and partly due to bottlenecks in other parts of the system.

The Blink version of GLGears uses SP register arithmetic to transform the gear objects independently of the client. To gauge the improvement obtained by not having to contact the client for each display update, we run two versions of Gears: GearsSP which uses stored procedure arithmetic and avoids context switching, and GearsSwitch which is updated by the client for each screen redraw. Figure 9.5 shows time-

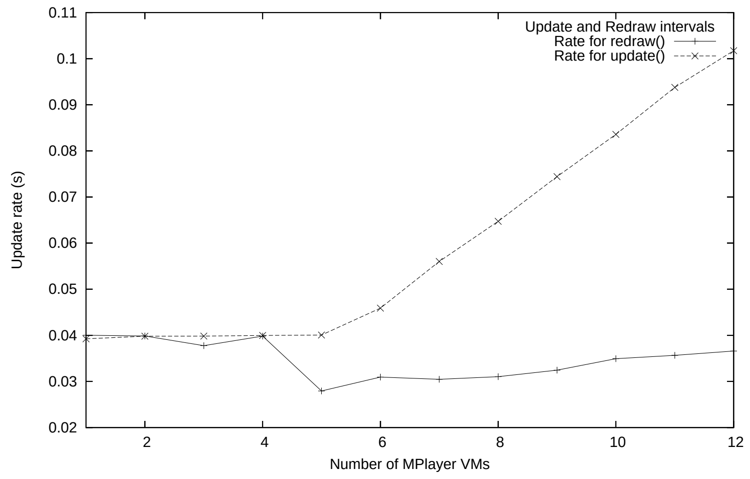


Figure 9.6: MPlayer performance on Blink.

Scenario	Execute
Native ATI driver call (gcc 3.3.6)	552 cpi
Blink Stored Procedure	554 cpi

Table 9.3: Cycles per OpenGL command, when executed using the native driver, or through JIT compiled code.

deltas as a function of the number of VMs. We see that GearsSP is able to maintain a steady frame rate of 50 frames per second for more than three times the amount of VMs than GearsSwitch, due to its avoidance of CPU context switches.

The final test was run on more modern hardware, a 2.1GHz Intel Core Duo 2 CPU, equipped with a 256MB nVidia GeForce 7900GS graphics card, and nVidia's proprietary graphics driver. Again, we are using the GLGears OpenGL demo as our benchmark, measuring total frames per second. Results are tabulated in table 9.4. In line with our previous results, BlinkGL stored procedures execute at close to native speed, while the overhead of interpreting command streams and off-screen buffer rendering results in approximately 25% drop in frames per second, for the OpenGL emulation case. We attribute most of this overhead to the extra copying needed for the off-screen buffer.

9.3 Self-Checkpointing in Blink

Blink allows an untrusted application to drive a 3D display, and allows the application to be implemented in a stateless manner. We are currently working on combining Blink with the self-migration and checkpointing mechanism. This combination allows us to live-checkpoint Blink VMs to stable storage, *e.g.*, to a hard- or flash drive, and to

Type of input	Redraw rate
Linux Direct Rendering	6834 fps
BlinkGL JIT compiled	6564 fps
Interpreted OpenGL on Blink	4940 fps

Table 9.4: Frames-per-second, for a 512x512 GLGears demo, on an Intel Core Duo 2, with a 256MB nVidia GeForce 7900GS graphics card.

fork and live-migrate application VMs across the network. This is similar to the system proposed by Potter and Nieh [2005], but with the difference that checkpointing is performed from *within* the VM, and may be done in the background without interrupting the user.

To test this in practice, we installed the Mozilla Firefox web browser in a Xen VM with 256MB of memory, on the nVidia-equipped 2.1GHz machine. The root file system was mounted as a read-only image, and directories such as `/var` and `/tmp` mounted as writable, memory-backed RAM disks on top. Inside the VM, we ran X11 on top of the Blink frame-buffer driver, described in section 9.1.6. We pointed the browser to the popular YouTube.com video-sharing site, and browsed through the selection of videos, while repeatedly checkpointing the VM to disk. The entire VM could be checkpointed to disk in less than 7 seconds on average, and resumed from disk in less than 6 seconds. During checkpointing, the video playing through YouTube's Flash plugin maintained a steady frame-rate.

Finally, we attempted to migrate the browser VM from the nVidia machine to an IBM Thinkpad T42p with an ATI Mobility FireGL graphics card. Despite differences in hardware, the VM was able to quickly resume on the target machine, using the display there. Migrating the 256MB VM showing a Youtube video, over a Gigabit Ethernet network, took roughly 15 seconds.

These results indicate that live self-checkpointing and migration is feasible, even in the context of user-facing graphical applications, and that Blink succeeds in providing a hardware-independent, restartable, display abstraction.

9.4 Chapter Summary

Blink demonstrates that today's advanced display devices can be multiplexed in a safe manner without poor performance. Blink emphasizes achieving safety with high performance, since enforcement overhead is often the main obstacle to the adoption of security mechanisms. In particular, a less efficient enforcement mechanism might not scale with the currently rapid growth in GPU capabilities.

Blink achieves safety by using a simple, fast JIT compiler and a shared-memory protocol that also helps reduce the cost of client/server communication. Blink further reduces overhead by amortizing JIT translation costs over multiple display updates and composing multiple applications to the same screen without a need for off-screen buffers. In addition, Blink remains backwards-compatible by employing an efficient and novel batch RPC interpretation technique, knows as Stack Replay. As a result, Blink allows for safe, practical display sharing even between soft-realtime graphics applications and legacy code.

Part III

Discussion

CHAPTER 10

Related Work

This chapter surveys classic works related to process migration, and recent work dealing with VM migration and other forms of distributed system management based on VMs or similar sandboxing techniques. For a more thorough survey of early process migration systems, the reader is referred to Milojicic et al. [2000].

10.1 Process Migration

Process migration systems were a hot topic in 1980s systems research, as many research operating systems evolved into distributed operating systems, and added the ability to preempt and relocate running applications. Examples include Distributed V [Theimer et al., 1985], Emerald [Jul et al., 1988], Sprite [Douglis and Ousterhout, 1991] and MOSIX [Barak and La'adan, 1998].

Though many distributed operating system research projects have had great impact, *e.g.*, distributed file systems such as AFS, this has not been the case with process migration. The survey by Milojicic et al. [2000] lists a number of possible reasons, including social factors, such as workstation users being unwilling to share their resources with others, and technical reasons such as residual dependencies and the requirement for homogeneity across hosting nodes, and the competition from simpler remote execution systems.

Residual dependencies arise when a migrated process still retains dependencies on the machine from where it migrated. Examples could be open file descriptors, shared memory segments, and other local resources. Such residual dependencies are undesirable both because the original machine must remain up and available, and because they usually negatively impact the performance of migrated processes, and may even entirely prevent the migration of certain types of processes.

For example Sprite processes executing on foreign nodes require some system calls to be forwarded to the home node for execution, leading to at best reduced performance and at worst widespread failure if the home node is unavailable. Although various efforts were made to ameliorate performance issues, the underlying reliance on the availability of the home node could not be avoided. A similar fragility occurs with MOSIX where a "deputy" process on the home node must remain available to support remote execution.

Zap [Osman et al., 2002] uses partial OS virtualization to allow the migration of process domains (pods), essentially process groups, using a modified Linux kernel. The approach is to isolate all process-to-kernel interfaces, such as file handles and sockets, into a contained namespace that can be migrated.

Most process migration systems require source and destination machine to be compatible in terms processor architecture and model, and this need for host homogeneity limits the set of hosts to which a process can be migrated. This problem was solved in Emerald [Steensgaard and Jul, 1995], by representing program execution state as platform-independent "bus stops", that could be converted to and from machine-specific representations. The program itself was compiled for each platform ahead of time.

Two current technology trends combine to make process migration systems more viable; the advent of massive compute clusters deployed at research institutions and powering search engines such as Google and MSN, and the growing interest in virtual machines. Clusters are less affected by social factors, are often very homogeneous, and VMs are independent and thus often free from residual dependency problems.

10.2 Persistent Systems

Previous work on self-checkpointing, or persistent, operating systems includes the EROS single-level store [Shapiro and Adams, 2002]. The main differences from our work are that with relatively few changes, we are able to retrofit a self-checkpointing facility onto a commodity operating system, and that we allow the checkpoint data to be passed through a user-space application for greater flexibility. The same low-level mechanism is used for implementing live migration, checkpointing, and network forks.

10.3 Virtual Machine Migration

10.3.1 The Collective

The Collective project [Sapuntzakis et al., 2002] has previously explored VM migration as a tool for providing mobility to users who work on different physical hosts at different times, citing as an example the transfer of an OS instance to a home computer while a user drives home from work. This work aimed to optimize for slow ADSL links and longer time spans, and stopped OS execution for the duration of the transfer, with a set of enhancements to reduce the transmitted image size.

10.3.2 VMWare VMotion

VMWare is a Virtual Machine Monitor (VMM) for the Intel Pentium architecture. It allows several operating systems to share a single machine. VMWare lets unprivileged code run directly on the CPU, at full speed, while interpreting privileged instructions, to trick the guest operating system into believing it is running directly on the real hardware. VMWare is able to host unmodified operating systems, though this transparency comes at the price of some performance degradation, compared to para-virtualized systems. Recent versions of VMWare ESX Server include "VMotion" functionality for VM live-migration as well [Nelson et al., 2005].

10.3.3 Internet Suspend/Resume

Kozuch and Satyanarayanan [Kozuch and Satyanarayanan, 2002] suggested the use of virtual machines (based on VMWare) as a kind of laptop replacement, for the Internet Suspend/Resume project. Their system stores suspended VMM images in a network or distributed filesystem, to make them accessible from multiple locations. More recent versions [Kozuch et al., 2004] split the file into 256kB chunks, placed in separate files, to allow the CODA distributed file system to track different versions of these chunks, and to pre-fetch ("hoard") chunks at frequently visited hosts. The ISR project focused mainly on end-user and WAN scenarios, the lowest reported downtime at migration is 45 seconds.

10.3.4 NomadBIOS

NomadBIOS is an application for the L4 microkernel. NomadBIOS runs as a user level application under L4, providing multiple NomadLinux instances with a VMM-style service layer, including virtualized address spaces and abstracted hardware-access. NomadLinux is a version of L4Linux, adapted to run entirely on top of the IPC abstractions provided by NomadBIOS, and with the ability to suspend its entire state to memory. NomadBIOS includes its own set of drivers and a TCP/IP stack, and was the first system to implement pre-copy live-migration of operating systems, with best-case freeze-times in the $10\ ms$ range.

10.3.5 μ-Denali

Denali was a specialized hypervisor for hosting of Internet services. The first Denali implementation did not virtualize the MMU, and was only able to host a specially crafted "Ilwaco" guest OS. A later version, known as μ-Denali [Whitaker et al., 2004b], included a virtual MMU, and also the ability to perform non-live "stop-and-copy" VM migration a para-virtualized NetBSD guest operating system.

Being virtual machine monitors, all of the above mentioned systems provide stronger isolation than traditional process migration systems, but from a security perspective, all suffer from the problem of having a complex, network-facing control interface.

10.4 VM Migration for Cluster Management

Migration is becoming a popular feature of virtual machine systems, and recent work has explored the use of migration for automated systems management. SandPiper [Wood et al., 2007] uses Xen live migration for hotspot mitigation, and the VMWare Distributed Resource Scheduler (DRS) is used for load balancing in production data centers. Similar techniques should be applicable to our platform.

The Shirako [Grit et al., 2006] project from Duke University has also explored the use of Xen VMs as the foundation for utility and grid computing. Similar to our token system, Shirako uses leases (implemented using SHARP cryptographic operations) for negotiating access to cluster resources, but Shirako's leases also support more advanced reservations, such as access to a group of machines. Shirako has been extended with support for live VM migration, to adapt to changes in resource demands or hardware availability.

Our Evil Man token system is simpler than the tickets and leases provided by SHARP. The benefit of our system is that it is simple enough that it can be implemented in a few hundred lines of trusted C code, and allows payment from the first network packet, preventing Denial-of-Service attacks such as TCP-SYN flooding. In addition, the small value of our tokens (enabled by cheap token generation and verification) reduces the initial investment needed when instantiating a new job-VM, and thus the risk of customer complains and need for human intervention. For complex federated systems, our low-level token system could be used in combination with a higher-level reservation system such as the one provided by SHARP.

10.5 Active Networks

The emergence of safe and platform-independent run-times such as the Java Virtual Machine [Lindholm and Yellin, 1996], coupled with the growth of wide-area networks such as the Internet, led to a surge in research in the area of Active Networks [Wetherall, 1999]. The claim was that "active architectures permit a massive increase in the sophistication of the computation that is performed within the network." [Tennenhouse and Wetherall, 1996], and that the ability to rapidly deploy code at network routers and switches would allow new forms of application multicast, and the enforcement of new network policies.

Like our work, this work depended on the ability to run untrusted code in a sandboxed environment. Unfortunately, the idea of letting strangers change the programming of expensive routing equipment used in production has not had much levy with Internet providers, but the idea of being able to run custom code on nodes scattered across the Internet has since evolved into systems such as PlanetLab [Peterson et al., 2006]. PlanetLab is a collection of PCs installed a research institutions across the globe. PlanetLab allows researchers to experiment with new services and overlay network under realistic network conditions. PlanetLab is based on Linux *VServer* [Soltesz et al., 2007] containers, a lightweight isolation mechanism that is similar to FreeBSD's "jails" feature [Kamp and Watson, 2000].

10.6 Micro-payments and Remote Capabilities

10.6.1 PayWord

Financial incentives in computing systems has been explored in systems such as Spawn [Waldspurger et al., 1992], SHARP [Fu et al., 2003], and Tycoon [Lai et al., 2004], and in older work as surveyed by McKell et al. [1979]. There seems to be a consensus that the ability to charge for resource access leads to better utilization, in addition to helping fund equipment and operational expenses. Most of the proposed systems use some form of auctioning for automatic pricing of resources. While few pay-for-use systems have been deployed for high performance computing, similar algorithms are used when selling advertising space on web sites, *e.g.*, by the Google Adwords system.

The Evil Man token system is similar to the PayWord [Rivest and Shamir, 1996] micropayment system. However, it is simplified even further by the use of shared-secret HMAC for initial token-chain signature verification, rather than full-blown public key signatures. In PayWord, the responsibility of token chain creation lies with the customer, whereas in Evil Man token chains are generated by the vendor and only released in response to continued payment for use. This removes problems of token chain revocation and return change, but increases the load on the token vendor.

10.6.2 PlanetLab Remote Capabilities

Evil Man boot tokens also have similarities to to the *rcap* remote capabilities used in the PlanetLab OS [Bavier et al., 2004], in that both specify limits on resource usage. Evil Man tokens are short-lived, requiring a constant supply of new tokens for the resource reservation to stay active. We believe this will lead to more fine-grained time division of resources, because there is less commitment to stay on a particular host. Because Evil Man uses Xen VMs rather than the lighter weight Linux VServers used in Planetlab OS, Evil Man is more flexible and more secure against kernel level security exploits, at the cost of using more resources.

10.6.3 Kerberos

Our token system also has similarities to Kerberos [Miller et al., 1987], where access control is centralized at a single host (known as the key distribution center), and temporary keys used for granting access to remote resources. For installations with existing Kerberos infrastructures, it is conceivable that our token vendor service could be "kerberized", and issue tokens based on Kerberos credentials.

10.7 Grid and Public Access Computing

10.7.1 Grid Systems

The Grid Computing [Foster et al., 2002] concept has received a lot of attention, and a number of different grid systems have been built or proposed. Common to most of the work on grid computing is that it builds on existing operating systems, and as such is limited to solutions that can be implemented as user-space applications [Karlsen and Vinter, 2006]. Because of the huge amount of research in this area, we limit the discussion here to systems that attempt to implement grid computing support at the operating system level, where the freedom to experiment is larger. Like us, Figueiredo et al. [2003] have also proposed the use of VMs as job containers from grid computing, as have Ruth et al. [2006], who also employ live migration of VMs and groups of VMs across hosts and institutions. Mirtchovski et al. [2004] have proposed the use of the Plan 9 operating system as a foundation for grid computing, claiming that Plan 9 is more suited to a networked environment than UNIX (and thus Linux), because UNIX was designed long before computer networks became pervasive. While we agree that grid computing needs OS support, and that the use of VMs will provide more flexible and robust run time environments, our work differs by its strong focus on control plane simplicity.

10.7.2 XenoServers

XenoServers [Hand et al., 2003] is another system that aims to build a public-access infrastructure based on Xen virtual machines. The platform has comprehensive support for wide-area resource discovery and negotiation, but to our knowledge does not attempt to minimize or reduce the trusted base install on participating nodes. Also, similar to our first prototype, it apparently does not address the problem of controlling NAT-hidden nodes.

10.7.3 BOINC

While our work has been focusing mostly on dedicated compute clusters, several projects have been exploring the use of volunteer compute resources. BOINC (Berkeley Open Infrastructure for Network Computing) is one such example, where users dedicate spare CPU cycles through a "screen-saver" client. Anderson [2004] mentions the problem of checkpointing, to prevent loss of state due to a client crash, but the BOINC middleware only reminds the application to checkpoint at regular intervals, it does not have the ability to ensure that checkpointing will actually happen, or that it will be correctly implemented. Because volunteer resources cannot be trusted, BOINC supports redundant processing, where each job is executed multiple times and results compared. In case of disagreement, new jobs are spawned until a quorom is reached. BOINC does not support job migration. BOINC executes code natively in the host operating system (Windows, Linux, MacOSX), and does not provide protection other than was is afforded by the host OS.

10.8 Trusted Display Systems

The X Window System (X11) facilitates remote display, and the protocol's extensibility means that other protocols such as OpenGL can be tunneled inside X connections. Unfortunately, currently popular versions of X have very large code bases, making them hard to trust security-wise. Efforts to create trusted X implementations [Epstein et al., 1993] have not had lasting impact, and many of the assumptions on which X is based (*e.g.*, the need for remote font servers or support for monochrome or color-mapped displays) are no longer relevant. For these reasons, we have chosen not to base our work on X.

Recently, the VMGL [Lagar-Cavilla et al., 2007] project has adopted Chromium [Humphreys et al., 2002] to work across VM boundaries, over TCP/IP. Like Blink, VMGL supports state tracking for use in VM checkpointing and migration, and VMGL currently implements a larger subset of OpenGL than Blink. Blink employs JIT compilation and static verification of stored procedures, and saves the overhead of passing all data through a pair of network protocol stacks by optimizing for the common case of client and server being on the same physical machine. VMWare Inc. has also announced experimental support for Direct3D virtualization in their desktop products.

Specialized secure 2D display systems for microkernels have been described for L4 [Feske and Helmuth, 2005] and EROS [Shapiro et al., 2004]. Both systems make use of shared memory graphics buffers between client and server, and both describe mechanisms for secure labeling of window contents. Our work addresses the growing need for 3D acceleration but currently our labeling mechanism is rather crude.

CHAPTER 11

Discussion and Conclusion

The ability to live-migrate applications between physical hosts is of increasing importance. The popularity of VMs for consolidation of many workloads onto fewer machines makes scheduling of downtime, for hardware or system software upgrades, hard. Migration solves this problem by allowing applications to be evacuated from a machine before taking it down for maintenance. Live VM migration cannot replace application specific fault-tolerance measures, but because VM migration has the ability to migrate *all* state, including open network connections, it will frequently provide an end-user experience superior to that of application-level fault-tolerance, in cases where downtime can be predicted. Live VM migration also allows a system administrator, or an automated scheduling service, to load-balance a data center or computing cluster. When we first proposed live VM migration with NomadBIOS in 2002, it could be described as a solution in search of a problem. Today, VMs are everywhere, and live VM migration is used routinely in corporate data centers. The question is no longer "if" but rather "how" live migration should be implemented.

11.1 Contributions

This dissertation makes the following contributions:

- The design of the self-migration algorithm.
- The implementation of self-migration in the Linux 2.4 and 2.6 operating systems.
- The design of a simple token system that lets untrusted users pay for resource access.
- The design and implementation of a minimal network control plane for on-demand cluster computing nodes, consisting of less than 400 lines of C code.
- The implementation of the Evil Man and Evil Twin system prototypes, for grid and on-demand computing.
- Experimental validation of the self-migration and control plane systems.
- The design and implementation of the "Blink" trusted display system, including a JIT compiler and an efficient interpreter for a superset of OpenGL.

- The design and implementation of the "Pulse" event notification system for simple, secure, and scalable wide-area cache invalidation.

With NomadBIOS, we showed the feasibility of live VM migration, by implementing a hypervisor supporting hosted VM migration. Some of the lessons we learned from that work were how the addition of hosted migration inflated the trusted software base, and that implementing migration at the lowest layer came at a cost of reduced flexibility, because the migration algorithm and security parameters had to be hard-coded into the base install.

In order to solve these problems, this dissertation describes the self-migration algorithm, that achieves live VM migration without host support. To our knowledge, we are the first to describe a way of incrementally checkpointing a running commodity operating system from within. The self-migration algorithm is useful not only for migration, but also for checkpointing a running system to stable storage, without noticeable interruption. Our work builds on our previous results with NomadBIOS, results that have been successfully transferred into the widely deployed Xen VMM. Through collaboration in the Xen project, we have shown the effectiveness of pre-copy migration for entire operating systems running production workloads. Through our work on self-migration and remote VM instantiation and bootstrap, we have shown that migration does not have to be provided as a trusted system service, but can be effectively implemented entirely at the application level.

To solve the problems of authentication and authorization, we have developed an electronic currency, or token system, to pay for access to resources. With this system, access control can be centralized at the token vendor, and the problem of stale or "zombie" domains is solved, because a system for which payment is not provided will converge towards quiescence, killing zombie domains as payments run out. In a large system with multiple competing resource-providers and customers, the Evil Man token system may be inadequate, but our intention has not been to build a complete resource economy, only to provide a basic mechanism and digital currency that allows charging for computing resources. The token system does not require the customer to negotiate a service contract with the individual host, and has no need for a judicial system to handle disputes in case of host failure or lack of service. The Evil Man token system can be used as a basis for construction of more elaborate systems, though actually doing so remains future work.

We make the observation that today's applications are too complex for traditional process models, and that the introduction of a VM-like container model will be of benefit, not only for server consolidation, but also in desktop computing. To that end, we design and implement a high-performance display system for desktop VMs, in a way that can be combined with self-checkpointing, to enable persistence of each VM application. For applications that are extended with third-party plugins, self-checkpointing is a robust way of implementing persistence at the application level.

A final contribution is the design and implementation of a simple, secure, and scalable system for wide-area cache-invalidation, that allows us to control the operation of a large number of compute nodes, even when some of them are hidden behind firewalls or NATs. The Pulse event notification mechanism is useful not only for distributed workload control, but also in other situations where data is replicated from a centralized source. The author gets all of his email through Pulse, and Pulse events are also used for rapidly deploying updates to the base install of our development machines and cluster nodes. Pulse would also be useful for RSS feeds and web logs that currently rely on periodic or manual polling, and we expect to release it as open source in the near future.

11.2 Design Alternatives

In our implementation of self-migration in Linux, we have tried to make use of existing Linux abstractions where possible. Page tables are accessed through existing macros and routines, and the checkpoint data passes through a standard Linux user-space application, using normal sockets or file handles. Where possible, memory is obtained through the Linux kernel allocator, and we use the Linux TCP/IP stack for network communication. These design choices increase our chances of getting the implementation adopted into mainline Linux, but also make it the implementation less portable to other VM guest operating systems. They also require our implementation to be continually kept up to date with upstream changes from the Linux community.

An alternative approach would be to access the underlying VMM and hardware abstractions in a more direct manner, and use an embedded TCP/IP stack such as UIP for network communication. Such an implementation would be less portable across different VMMs, but would be easier to port to other guest operating systems, *e.g.*, to Windows.

A final alternative that is of interest in purely virtualized scenarios, *i.e.*, when the guest OS cannot be modified to run under the VMM, would be an implementation in a separate, unprivileged VM. This VM could also provide emulation of legacy hardware, bridging a guest OS with legacy drivers to virtualized of direct-I/O drivers, and with the ability to live-migrate the guest at will. Such a *stackable virtual machine* [Ford et al., 1996] approach would allow for live migration of unmodified, legacy guests, without adding undue complexity to the privileged, trusted pars of the VMM.

11.3 Why Transparency is not Always a Good Thing

Most virtual machine research in recent years has focused on adding new functionality to the VMM, transparently to the hosted applications. Examples include memory-sharing [Waldspurger, 2002], fault-tolerance through deterministic replay [Bressoud and Schneider, 1996], configuration management [Whitaker et al., 2004a], intrusion detection [Dunlap et al., 2002], and mobility [Sapuntzakis et al., 2002]. Our work is different, in that we generally advocate doing things inside the VM, rather than in the VMM or host operating system. While we recognize that there are things that can only be done efficiently at the VMM level, in our opinion, the real benefit of a VMM over a traditional OS lies in its simplicity. If new features are continuously being added to the VMM or other trusted components, the VMM ends up becoming just another operating system—and it is conceivable that at some point we will need a new abstraction to go *below* the VMM to repair its shortcomings.

With our work we have attempted to show how transparent solutions not always are the only option. The self-migration algorithm demonstrates that functionality that would seemingly require a VMM, can be implemented completely without VMM or host OS assistance. The Blink display system cannot function without a trusted component, but the approach shows that most of the functionality, *e.g.*, OpenGL or X11 emulation, can be implemented inside the VM, instead of in the VMM.

Technology trends do not always favor transparent solutions. For instance, current VMM systems add I/O overhead, and the solution is to extend the hardware to allow VMs to control their I/O devices without involving the VMM. Unfortunately, this breaks transparent migration, because the state of the VM can now change without the VMM's knowledge. Self-migration will still work, as long as the driver for the direct I/O device cooperates in logging page changes.

11.4 Conclusion

Compared to traditional operating system processes, virtual machines offer superior functionality. VMs are able to control their own task-scheduling, control their own paging and virtual memory layout, and to handle their own bootstrap. We have shown how VMs can also self-checkpoint themselves, to a another host for migration, or to stable storage. Our *self-migration* implementation does not require any functionality to be added to the VMM or to other parts of the trusted computing base, and experimentally confirms the thesis stated in chapter 1. The implementation is simple, adding less than 2000 lines of code to the targeted guest operating system.

Part IV

Appendices

APPENDIX A

Bibliography

Adams, K. and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, 2006.

Anderson, DP. BOINC: a system for public-resource computing and storage. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10, 2004.

Annapureddy, Siddhartha, Michael J. Freedman and David Mazi&#232;res. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

Audet, F. and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), 2007.

Barak, A. and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, 1998.

Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

Bavier, Andy, Larry Peterson, Mike Wawrzoniak, Scott Karlin, Tammo Spalink, Timothy Roscoe, David Culler, Brent Chun and Mic Bowman. Operating system support for planetary-scale network services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

Bellare, M., R. Canetti and H. Krawczyk. Keying hash functions for message authentication. In Koblitz, N., editor, *Advances in Cryptology - Crypto 96 Proceedings*, volume 1109. Springer-Verlag, 1996.

Bensoussan, A., C. T. Clingen and R. C. Daley. The multics virtual memory: concepts and design. *Commun. ACM*, 15(5):308–318, 1972.

Bershad, B. N., S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

Birrell, Andrew D. and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

Black, Andrew P. Supporting distributed applications: experience with eden. In *Proceedings of the Tenth ACM symposium on Operating Systems Principles*, pages 181–193. ACM Press, 1985.

Blythe, David. The Direct3D 10 System. In *Proc. of the 33rd ACM SIGGRAPH conference*, 2006.

Bressoud, Thomas C. and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

Buck, I. G. Humphreys and P. Hanrahan. Tracking graphics state for networked rendering. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 87–95, 2000.

Bugnion, Edouard, Scott Devine, Kinshuk Govil and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

Burns, Greg, Raja Daoud and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

Burrows, M. The Chubby lock service for loosely coupled distributed systems. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI-02)*, 2006.

Cabrera, L.F., M.B. Jones and M. Theimer. Herald: Achieving a global event notification service. *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 87–94, 2001.

Castro, M., P. Druschel, A.M. Kermarrec and AIT Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

Chandra, R. N. Zeldovich, C. Sapuntzakis and M.S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, pages 259–272, 2005.

Clark, Christopher, Keir Fraser, Steven Hand, Jacob G. Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Networked Systems Design and Implementation NSDI '05*, 2005.

Cox, Richard S. Jacob G. Hansen, Steven D. Gribble and Henry M. Levy. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

Creasy, R. J. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

De Winter, D., P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt and P. Demeester. A Hybrid Thin-Client protocol for Multimedia Streaming and Interactive Gaming Applications. In *the 16th Annual International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2006.

Douglis, Fred and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21 (8):757–785, 1991.

Dunkels, Adam. Full tcp/ip for 8-bit architectures. In *Proceedings of the first international conference on mobile applications, systems and services (MOBISYS 2003)*, 2003.

Dunlap, George W., Samuel T. King, Sukru Cinar, Murtaza A. Basrai and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

Epstein, J., J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Dancer, C.R. Martin, M. Branstad, G. Benson and D. Rothnie. A high-assurance window system prototype. *Journal of Computer Security*, 2(2-3):159–190, 1993.

Erlingsson, U. *The Inlined Reference Monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2004.

Erlingsson, U., M. Abadi, M. Vrable, M. Budiu and G.C. Necula. XFI: Software guards for system address spaces. In *In Proceedingss of the 7th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

Ertl, M. and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters, 2003.

Faith, Rickard E. and Kevin E. Martin. A security analysis of the direct rendering infrastructure, 1999. http://precisioninsight.com/dr/security.html (archive.org copy).

Feske, N. and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *In Proceedings of the 21st Annual IEEE Computer Security Applications Conference*, pages 85–94, 2005.

Figueiredo, R. J., P. A. Dinda and J. A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550. IEEE Computer Society, 2003.

Ford, Bryan, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back and Stephen Clawson. Microkernels meet recursive virtual machines. *SIGOPS Oper. Syst. Rev.*, 30(SI):137–151, 1996.

Foster, Ian, Carl Kesselman, Jeffrey M. Nick and Steven Tuecke. The Physiology of the Grid. An open Grid services architecture for distributed systems integration. draft., 2002.

Fraser, Keir, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, 2004.

Fu, Yun, Jeffrey Chase, Brent Chun, Stephen Schwab and Amin Vahdat. Sharp: an architecture for secure resource peering. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148, New York, NY, USA, 2003. ACM Press.

Fuller, V. and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632 (Best Current Practice), 2006.

Goldberg, Robert P. Survey of virtual machine research. *IEEE Computer*, 7(6):34–45, 1974.

Grit, L., D. Irwin, A. Yumerefendi and J. Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration. In *Proceedings of First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, 2006.

Hand, S., T. Harris, E. Kotsovinos and I. Pratt. Controlling the XenoServer Open Platform. *Open Architectures and Network Programming, 2003 IEEE Conference on*, pages 3–11, 2003.

Hand, Steven M. Self-paging in the nemesis operating system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 73–86. USENIX Association, 1999.

Hansen, Jacob G. Blink: Advanced display multiplexing for virtualized applications. In *Proceedings of the 17th International workshop on Network and Operating Systems Support for Digital Audio & Video (NOSSDAV)*, 2007.

Hansen, Jacob G., Eske Christiansen and Eric Jul. The laundromat model for autonomic cluster computing. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, 2006.

Hansen, Jacob G., Eske Christiansen and Eric Jul. Evil twins: Two models for tcb reduction in clusters. *Operating Systems Review, special issue on Small Kernel Systems*, 2007.

Hansen, Jacob G. and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

Hansen, Jacob G. and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

Hansen, Jacob G. and Eric Jul. Optimizing Grid Application Setup using Operating System Mobility. *Lecture Notes in Computer Science*, 3470:952, 2005.

Hansen, Per Brinch. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–250, 1970.

Härtig, Hermann, Michael Hohmuth, Jochen Liedtke and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.

Howard, John H., Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

Humphreys, G. M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner and J.T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

Hunt, Galen C. and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

Jul, Eric, Henry Levy, Norman Hutchinson and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

Kaashoek, M. Frans, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, New York, NY, USA, 1997. ACM Press.

Kamp, Poul-Henning and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings, SANE 2000 Conference*, 2000.

Karlsen, Henrik Hoey and Brian Vinter. Vgrids as an implementation of virtual organizations in grid computing. *wetice*, 0:175–180, 2006.

Kent, S. and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401

(Proposed Standard), 1998. Obsoleted by RFC 4301, updated by RFC 3168.

Kilgard, Mark J.˙ Realizing OpenGL: two implementations of one architecture. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 45–55, New York, NY, USA, 1997. ACM Press.

Kozuch, M. and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

Kozuch, Michael, Mahadev Satyanarayanan, Thomas Bressoud, Casey Helfrich and Shafeeq Sinnamohideen. Seamless mobile computing on fixed infrastructure. *Computer*, 37(7):65–72, 2004.

Lagar-Cavilla, H. Andres, Niraj Tolia, Mahadev Satyanarayanan and Eyal de Lara. VMM-independent graphics acceleration. In *Proceedings of VEE 2007*. ACM Press, 2007.

Lai, Kevin, Bernardo A. Huberman and Leslie Fine. Tycoon: A Distributed Market-based Resource Allocation System. Technical Report arXiv:cs.DC/0404013, HP Labs, Palo Alto, CA, USA, 2004.

LeVasseur, J., V. Uhlig, J. Stoess and S. Gotz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2004.

Levy, Henry M. and Peter H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, 1982.

Liedtke, Jochen. Improved Address-space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces. Technical report, GMD-Forschungszentrum Informationstechnik, 1995.

Liedtke, Jochen. Microkernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, 1996.

Lindholm, Tim and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

Lougheed, K. and Y. Rekhter. Border Gateway Protocol 3 (BGP-3). RFC 1267 (Historic), 1991.

Lowell, D.E., Y. Saito and E.J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 211–223, 2004.

Matthews, Jeanna Neefe. The case for repeated research in operating systems. *SIGOPS Oper. Syst. Rev.*, 38(2):5–7, 2004.

McKell, Lynn J., James V. Hansen and Lester E. Heitger. Charging for computing resources. *ACM Comput. Surv.*, 11(2):105–120, 1979.

Miller, S. P., B. C. Neuman, J. I. Schiller and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, Massachusetts Institute of Technology, 1987.

Milojicic, D., F. Douglis, Y. Paindaveine, R. Wheeler and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

Mirtchovski, Andrey, Rob Simmonds and Ron Minnich. Plan 9 - an integrated approach to grid computing. *ipdps*, 18:273a, 2004.

Motta, Giovanni, James Gustafson and Samson Chen. Differential compression of executable code. *dcc*, 0:103–112, 2007.

Nath, Partho, Michael Kozuch, David O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX '06) (to appear)*, Boston, MA, 2006.

Nelson, Michael, Beng-Hong Lim and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, 2005.

Noack, Mathias. Comparative Evaluation of Process Migration Algorithms. Master's thesis, Techical University of Dresden, 2003.

NSFISSI, . National information systems security (infosec) glossary, 1997. NSFISSI No.4009.

Osman, S., D. Subhraveti, G. Su and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI-02)*, pages 361–376, 2002.

Perkins, C. E. and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1997.

Peterson, L., A. Bavier, M.E. Fiuczynski and S. Muir. Experiences building planetlab. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

Pfaff, Ben, Tal Garfinkel and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *3rd Symposium of Networked Systems Design and Implementation (NSDI)*, 2006.

Popek, Gerald J. and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

Potter, Shaya and Jason Nieh. Webpod: persistent web browsing sessions with pocketable storage devices. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 603–612, New York, NY, USA, 2005. ACM Press.

Rashid, Richard, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

Ravn, A. P. Device monitors. *IEEE Transactions on Software Engineering*, 6(1):49–52, 1980.

Ritchie, Dennis M. and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

Rivest, Ronald L. and Adi Shamir. Payword and MicroMint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

Robin, John Scott and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 2001 USENIX Security Symposium*, 2001.

Ruth, Paul, Junghwan Rhee, Dongyan Xu, Rick Kennell and Sebastien Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing*, 2006.

Saltzer, J. H., D. P. Reed and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

Saltzer, Jerome H. and Michael D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 1974.

Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

Sapuntzakis, C. P., R. Chandra, B. Pfaff, J. Chow, M. S. Lam and M.Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, 2002.

Satran, J., K. Meth, C. Sapuntzakis, M. Chadalapaka and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Standards Track), 2004.

Shapiro, Jonathan S. and Jonathan Adams. Design evolution of the eros single-level store. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association.

Shapiro, Jonathan S., John Vanderburgh, Eric Northup and David Chizmadia. Design of the EROS trusted window system. In *Proceedings of the Thirteeenth USENIX Security Symposium*, San Diego, CA, 2004.

Shen, Vincent Y., Tze-Jie Yu, Stephen M. Thebaut and Lorri R. Paulsen. Identifying error-prone software—an empirical study. *IEEE Trans. Softw. Eng.*, 11(4): 317–324, 1985.

Skoglund, Espen, Christian Ceelen and Jochen Liedtke. Transparent orthogonal checkpointing through user-level pagers. *Lecture Notes in Computer Science*, 2135, 2001.

Snoeren, Alex C. and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM Press, 2000.

Soltesz, Stephen, Herbert Pötzl, Marc Fiuczynski, Andy Bavier and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of EuroSys 2007*, Lisbon, Portugal, 2007.

Steensgaard, B. and E. Jul. Object and native code thread mobility among heterogeneous computers (includes sources). *SIGOPS Oper. Syst. Rev.*, 29(5):68–77, 1995.

Sugerman, Jeremy, Ganesh Venkitachalam and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

Sundararaj, A. and P. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)*, 2004.

Swift, Michael M., Brian N. Bershad and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005.

Swift, M.M., M. Annamalai, B.N. Bershad and H.M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

Tanenbaum, Andrew S., Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, 1990.

Tennenhouse, David L. and David J. Wetherall. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev.*, 26(2):5–17, 1996.

Theimer, Marvin M., Keith A. Lantz and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.

Thomas, Rob and Jerry Martin. The Underground Economy: Priceless. *;login:*, 31(6), 2006.

U.S. Department of Commerce, . Secure hash standard (shs). Federal Information Processing Standards Publication 180-2, 2002.

Wahbe, Robert, Steven Lucco, Thomas E. Anderson and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM Press.

Waldspurger, C. A. Memory resource management in VMware ESX server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 181–194, Boston, MA, USA, 2002.

Waldspurger, Carl A., Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Trans. Softw. Eng.*, 18(2):103–117, 1992.

Wang, W. and R. Bunt. A self-tuning page cleaner for DB2. *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*, pages 81–89, 2002.

Warfield, A., R. Ross, K. Fraser, C. Limpach and S. Hand. Parallax: Managing Storage for a Million Machines. *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X), Santa Fe, NM, June*, 2005.

Wetherall, David. Active network vision and reality: lessions from a capsule-based system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 64–79, New York, NY, USA, 1999. ACM Press.

Whitaker, Andrew, Richard S. Cox and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 77–90. USENIX Association, 2004a.

Whitaker, Andrew, Richard S. Cox, Marianne Shaw and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004b.

Whitaker, Andrew, Marianne Shaw and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

Wood, T., P. Shenoy, A. Venkataramani and M. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the Fourth Symposium on Networked System Design and Implementation (NSDI '07)*, 2007.

Wright, C. P., J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, 2006.

Zayas, E. Attacking the process migration bottleneck. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24, New York, NY, USA, 1987. ACM Press.