Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Jeffrey A. Turkstra

Entitled
Metachory: An Unprivileged OS Kernel for General Purpose Distributed Computing

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

DAVID G. MEYER
Chair

SAMUEL P. MIDKIFF

CHARLIE HU

MARK C. JOHNSON

RICHARD L. KENNELL

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): DAVID G. MEYER

Approved by: V. Balakrishnan 04-17-2013
Head of the Graduate Program Date

METACHORY: AN UNPRIVILEGED OS KERNEL FOR GENERAL PURPOSE DISTRIBUTED COMPUTING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jeffrey A. Turkstra

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2013

Purdue University

West Lafayette, Indiana

I dedicate this dissertation to...

my wife, Shelley L. Turkstra. I would have given up long ago were it not for her—you are the love of my life and my strength;

my son, Jefferson E. Turkstra, the joy of my life and hope for the future;

my grandmother, Aleatha J. Turkstra, my connection to the past and source of limitless endurance.

ACKNOWLEDGMENTS

It is impossible to enumerate all of the people that have made this dissertation possible—my memory is just not that good. If I didn't mention you, and you feel like I should have, you're probably right, and I apologize.

First and foremost, I would like to thank my parents, Mr. Donald J. Turkstra and Ms. Susan E. Turkstra. You are responsible for my existence and shaping the core of my being. Without you, I would never have made it this far in life. I will never be able to fully appreciate your sacrifices, because you have made sure I will never have to make them myself. I also would like to thank my sister, Mrs. Sarah E. Bruischat, for the occasional distractions and having someone to talk to about anything. Ms. Susan Boesen (Jackson), my fourth grade teacher, you changed my path from leading to nowhere to leading, instead, somewhere amazing.

My committee members, Professor David G. Meyer, Dr. Mark C. Johnson, Professor Samuel Midkiff, Professor Y. Charlie Hu, and Dr. Richard L. Kennell. Dave, for supporting all of my endeavors. The flexibility that you gave me made this all possible. You are the single greatest teacher that I have ever had. Mark, you gave me a chance as an undergrad and without your support I wouldn't have even made it into graduate school. The opportunities you have afforded me at this university are priceless and your friendship and mentoring invaluable. Sam, someone whose perceptions align with mine surprisingly often, you forced me to put forth the extra effort that makes me feel like I've almost earned this degree. You too have been an invaluable mentor. Charlie, your support both with my research and my teaching activities has also been priceless. I wish that I could give you something in return. Finally, Rick, you and my wife are honestly the only two reasons that I'm graduating. You are an incredible friend, an amazing colleague, and an intellectual titan. There

is no possible way for me to thank you enough. You are simply awesome. I hope that you are able to find happiness among the rest of us.

Dr. Michael McLennan, my boss, your generosity has allowed me to comfortably balance graduate school and a family, not to mention experience the gloriousness of a season of skiing. There is no way to adequately express my gratitude. Thank you.

Dr. David A. Leppla, you set the standard for honor, integrity, and competence by which I judge myself, and I am indescribably grateful to have been able to share your final six years of tenure here at Purdue. The university will be a better place once all of the people responsible for your departure have gone.

Dr. Ryan D. Riley, my brother not in blood, but in bond, without you I probably wouldn't have made it through my undergraduate years. That said, and while you may have beat me to the end, I never switched teams. ;-)

Mr. Joshua C. Koon, my closest friend, thank you for the hours upon hours of interesting discussion and debate, listening to me when I am frustrated, and for being one of two people that I can talk with about virtually anything.

Mr. Jonathan E. Rexeisen, another close friend, thank you for being someone that I can always count on to test something, complain to, and debate my often hairbrained ideas.

Dr. Ahmed Amin, a good friend, office mate, business partner, and consumer of time that I lacked, I thoroughly enjoyed working on the microfluidics prototypes.

Mr. Matt Golden, the ECE graduate program administrator, I have never met a more patient human being. Thank you for always providing me with competent, quick help.

And last, but not least, I would like to thank the late Dr. Carl E. Sagan. This world needs men like him now more than ever. If my life has even 1/1024th of the impact on this world that his did, I shall consider myself incredibly successful.

"If I have seen further it is by standing on ye sholders of giants." — Isaac Newton

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Turkstra, Jeffrey A. Ph.D., Purdue University, May 2013. Metachory: An Unprivileged OS Kernel for General Purpose Distributed Computing. Major Professor: David G. Meyer.

Virtualization has provided a vehicle for people to gain flexibility and security in utilizing computing resources. At the same time, distributed systems have emerged to support large workloads and efficient use of clustered resources. Such distributed systems are often highly specialized and require considerable effort on behalf of the application developer, end-user, system administrator or some combination of all three to use. As a result, it is difficult, if not impossible, for ordinary end-users to take advantage of large-scale distributed computing.

To address these problems, we have created a new OS kernel that virtualizes only the exceptional aspects of native program execution such as system calls, page faults, signals, and file system interactions. Combined with a distributed coherence mechanism, our kernel proves to be a feasible approach to creating a general purpose distributed system that is not only simple to install, configure, and maintain but, more importantly, easy to use. Our approach provides a foundation for supporting execution of unmodified native applications—at nearly-native speed—by ordinary end-users without administrative or special privileges (e.g., "root") as well as transparent process migration and checkpointing.

1. INTRODUCTION

Distributed computing remains largely inaccessible to most users. This is primarily due to the rigidity of historical policies that were intended for deeply centralized infrastructure. These policies were based on assumptions that, at the time, were appropriate: computational resources were limited so allocation had to be strict and fair, storage was limited so it had to be centralized, trust models were homogeneous and absolute, applications were primitive, and the userbase was expected to be highly sophisticated.

It is often the case in industrial and academic venues that a researcher develops a computational model on a workstation and wants to run this model many times in a distributed fashion. Over 95% of the systems in the current Top 500 list run Linux or Unix [1], making these platforms a significant target for development. Nevertheless, even if the researcher's development workstation is the same platform as one of these large computational environments, he or she still faces several barriers to distributing an application. These barriers include setting up specialized kernels and runtime environments, recompiling or relinking applications, gaining authorization to access computational facilities, using a batch-oriented job submission system, and transmitting and recovering data associated with the application.

Today, we are surrounded by an abundance of computers, the most pedestrian of which would be regarded as pure fantasy at the time many of these policies were created. Unfortunately, this resource explosion has not been matched by a commensurate broadening of general utility for doing calculations. In an ocean of computational resources many who wish to use them are faced with the classic problem of, "Water, water everywhere and not a drop to drink."

This problem is compounded further by advances in many disciplines of science as well as ever-expanding simulation techniques. These advances have led to a new class

of scientific applications. These jobs are often relatively short running, but numerous in quantity. For example, when designing a new nanotube field effect transistor, experimentalists often generate a model and run simulations to get some idea of how the device will behave given certain characteristics and environmental conditions. One simple characteristic may be the environment's temperature. Determining electron and hole concentrations at a given temperature involves using the Finite-Element-Method of solving the Poisson Equation to generate a full three-dimensional electrostatics solution. Solution of this set of equations is computationally expensive, taking a single workstation many hours to solve. However, running the simulation on a cluster system can reduce this time down to minutes. Unfortunately, if the execution environment is queue-based, this time may expand back to hours or days.

Experimentalists increasingly desire to use sweeps of device parameters as well. That is, they wish to simultaneously run many simulations, checking to see if devices behave in a desired fashion. If, as the sweeps progress, it turns out that parameters require adjustment, the researcher would like to terminate the simulation mid-computation, without waiting for it to complete. This has resulted in the advent of more interactive simulation environments like science gateways [2] where the problem is compounded yet again. These environments run a new class of scientific applications that have requirements that are fundamentally at odds with a traditional queue-based resource manager:

1. Support for interactive applications

2. A need to execute them immediately

Traditional grid and cluster computers can provide limited interactivity through a "head node," but modern infrastructures like the TeraGrid/XSEDE [3] and the Open Science Grid [4] are increasingly discouraging this type of activity in favor of a fully automated staging and submission system. Such systems are, again, largely geared toward classic scientific computing applications. Other systems such as Condor [5] and GridEngine [6] provide support for checkpointing and migration and permit dynamic

introduction and withdrawal of computer resources. Nevertheless, these systems are still encumbered by a queue at the front-end and provide little support for interactive applications.

Modern scientific simulations require a new type of system—one that not only supports dynamic withdrawal and introduction of resources but that also permits dynamic and interactive use. In some cases, this will necessarily mean overcommitment of some resources in a way that slows down existing long-running applications for the sake of starting new quick ones. This goes against the well-understood ways to most efficiently use a computer resource for highest throughput. The justification is that users who wait on long-running applications can usually afford to wait a little longer. Furthermore, it is increasingly the case that researchers are willing to sacrifice throughput to decrease latency, avoid waiting in a queue, and allow a more interactive experience.

In the early mainframe era, the common mode of computing involved the submission of decks of punch cards, long waits, and inspection of print-outs. Often, a user would pick up a print-out for the sole purpose of determining how and why the submitted program failed to run. This was a highly-efficient, but very cumbersome approach to the use of limited resources. The arrival of remote terminals brought about an interactive revolution as individuals could not only access the computing resource remotely, but also do so with immediacy. We propose a system that facilitates the same type of revolution with high-performance computing—moving us from opaque, queue-based systems to interactive, dynamic systems.

In addressing the two requirements established above, we have identified six main technical challenges. Our system must:

1. map a user's data into remote computers without the need to manually copy it or create a shared file system

2. allow unmodified native interactive applications to run immediately rather than wait in a queue

3. allow a user to issue only applications to run—not have to create or maintain operating systems and software stacks

4. perform all operations on a conventional platform without the need for special hardware support, administrative privileges, or modification of the host OS

5. allow dynamic introduction and withdrawal of both applications and computational resources, employing transparent migration to balance load as well as vacate machines that are to be withdrawn

6. allow machine owners to offer their computers for use by others without risk

To address these challenges, we have built *Metachory*, or MX for short. Metachory constitutes a distributed OS kernel that runs entirely as a regular, unprivileged program. The most important aspects of this system are that it offers nearly-native speed of execution for unmodified native applications—incurring penalties only for data distribution—and that all elements of the system can be run by ordinary users with minimal effort. Resource providers are insulated from all actions of guest users by a virtualization layer.

The name Metachory is the latinization of the made-up Greek word "$\mu\epsilon\tau\alpha\chi\omega\rho\eta$" This word is composed of the prefix "meta-" meaning "beyond", "after", or "transcend" and the root "chory" (or "choric") meaning "space" or "place." Combining the two results in "Metachory" meaning roughly "transcendence of place."

The rest of the dissertation is organized as follows: In Chapter 2 we give an overview of the basic problems involved with distributing computation to expand on the rationale for the goals we outlined above. We clarify what we mean by "general purpose" distributed computing and our motivation in pursuing this work in Chapter 3. Chapter 4 shows the architecture of our distributed kernel while Chapter 5 illuminates our approach to distributed coherence. Chapter 6 covers the virtual file system and its implementation. Chapter 7 explains our method for process virtualization in detail. In Chapter 8 we discuss elements of process migration and how it works with

the distributed kernel. We show the results of performance evaluation of our system in Chapter 9. We summarize our discussion in Chapter 10 and outline remaining work.

2. BACKGROUND

Suppose that an ordinary, unsophisticated computer user wants to take the large collection of data accumulated on his or her personal computer and process it with some form of distributed computing. The approaches to accomplishing this task are numerous. By thoroughly exploring this problem and its constraints we can define what we mean by general purpose distributed computing as well as justify the motivation for our work.

2.1 Exporting a file system to the computing domain

If our user is part of a traditional academic or corporate environment he or she will likely be encouraged to use a large-scale computational cluster deployed and administered by an institutional computing support team. Such a solution is typically employed to maximize benefits to a large community of users while minimizing capital equipment costs. Nevertheless, there is usually a sharp division between the user's desktop computer and the cluster. Since there is no common file system between the two environments, the user must first transfer files from the desktop to the cluster. When calculations are complete the user must transfer files back to the desktop for further analysis. Most users are not permitted to extend their personal file system into the computational environment because:

1. Few if any network file systems can redistribute the contents of another distributed file system. For instance, popular cluster file systems such as GFS2 [7], GPFS [8], OCFS2 [9], and others [10–12] expect to export a block or object store—rather than another file system—to their clients. Common network file systems such as AFS [13] and others [14, 15] can export native file systems but typically cannot re-export another remote file system. The user would instead

have to serve data to each cluster member. Often this is made impossible due to the common practice of using private networks that allow connections only between cluster members, file servers and a few administrative access machines.

2. The ability to provide storage to a multi-user computational environment would require the provider to be responsible for restricting access to this data from the other users of the cluster.

3. There must be a consistent namespace to mount the file system on each cluster member and this might not be where the user would prefer it to be. As a pathological example, consider the situation if our user wanted his or her data to appear under /usr/mydata/ of a Linux cluster. This situation would only be made worse by two or more users who each independently wanted the same thing. There is typically no namespace encapsulation present in cluster environments that could keep these users separate.

4. Providing storage to a computer requires a high level of accountability to keep that service working in a reliable fashion. Often a failure of a network storage system accessed by only one user will cause the entire machine to become unusable. Inevitably, the support team would take the blame.

Even if serving data to a remote computational domain were possible it would still require the user in question to set up and maintain some sort of file server. This proposition would be accompanied by all of the usual concerns about security and access control that most individuals—especially our novice user—do not wish to consider. Ultimately there is no general way to satisfy challenge #1 in our list of challenges from Chapter 1.

2.2 Invocation of computation

Assuming the issue of data distribution is solved, the next question the user will face is how to start and manage the execution of computations. The approach taken

by an institutional support team is to deploy a conventional batch submission system such as LSF [16], PBS [17], Linda [18], Condor [19], BATRUN [20], or GridEngine [6] to let users queue operations from a trusted machine to cluster nodes. These solutions provide fair scheduling in the presence of multiple users, jobs and machines. They can also set up parallel execution environments and provide varying degrees of file staging. Such an environment is common for classical computer users who need to run many instances of a restricted set of applications and want to do so as efficiently as possible. It is often the case that such applications will additionally be run inside specialized execution environments [18,21–24] that require the application be relinked and changed in other ways to let it run in the specific target environment. Job submission is even less interactive and more abstracted for larger-scale computing environments such as those of grid systems [3, 4]. The user finds no fulfillment of challenge #2.

Consider the added constraint that the user wants to run one or more *interactive* applications where he or she can not only see the instantaneous output and progress of an application but adjust or terminate the application at will. Although batch queuing systems sometimes permit users to invoke interactive jobs they still require the end-user to wait for a slot in the scheduling queue and they last for a limited time—regardless of whether or not they are used for any computation. This means that the user must be ready when the queue slot becomes available. Time slots allocated for one or more machines are the customary unit of accounting for such environments.

Instead of having to wait for the computer, this user would rather invoke an application on whatever resources are available—overcommitting them if necessary. Rather than relying on efficient use of a time slot that can expire, the application accounting should happen only for resources consumed if and when the application is able to run. This also means that, for the sake of administrative flexibility, an application must be able to migrate or be checkpointed when its physical resources must go away or be pressed into service for some other more important application.

Condor [19] permits this kind of migration and accounting, but only for limited applications (e.g. no forking, no out-of-system communication with either an end-user or another application), and even then requires that applications be specially linked to run in this environment. Condor cannot migrate a native application, leaving challenge #5 only partially fulfilled.

2.3 Creation of computing environments

A lack of traditional computational resources has driven some researchers to find ways of creating large-scale computing environments for free with "volunteer computing" initiatives. Systems such as BOINC [25] (née SETI@home) allow organizations with a tremendous quantity of short computational jobs to outsource them to millions of personal computers over the Internet. To do so, they must build a specialized application that can be encapsulated in a distribution protocol. These applications are not interactive nor do they use any kind of shared file system. The applications have a limited and clearly defined input and output. Volunteers are reimbursed only in terms of goodwill, reducing the likelihood that they will try to forge or corrupt results. Nevertheless, results must be validated either by extra checks or redundant computation. Meanwhile, volunteers must trust the distributing organization not to do harmful things to their personal computers.

Volunteer computing environments, as they exist now, only make sense for organizations with a large backlog of computations. They must be willing to invest the time and effort of breaking those computations into small chunks so that they can be distributed. Those organizations must also be willing to wait for their turn in the distribution schedule before results begin to be generated. From the perspective of the person who wants to perform distributed computation, this system is operationally similar to conventional batch queuing systems, however the supply of resources is much higher. By virtue of the ease of introducing a computer to the system, BOINC

has become one the largest computing infrastructures in the world. This makes a compelling case for ease of use.

2.4 Distributed operating systems

Inflexibility of conventional job queuing systems have led some researchers to deploy and manage their own fully distributed operating systems. Over the years distributed systems such as Amoeba [26], MOSIX [27], Sprite [28, 29], V [30], and others [26, 31–34] were created to allow transparent distributed computing. From an end-user standpoint, this is ideal because it allows one to run any application as usual and the details of distribution and load balancing are handled entirely by the underlying distributed kernel.

The principle shortcoming of distributed kernels is that all machines must be homogeneous with respect to trust and administrative control. There is no provision for an untrusted individual to take an arbitrary machine and dynamically introduce it to or remove it from a distributed kernel environment. Since the kernel on each machine shares resources with kernels on other machines, anyone with low-level control of one machine can find ways to gain such control on another machine participating in the distributed kernel environment. Practically, this means it would not be possible to allow volunteers to join a machine with a distributed kernel environment in the same way that they could with BOINC. This limits the growth and, therefore, the utility of distributed kernel systems.

A secondary impediment to the adoption of distributed kernel systems is that each machine in question must run the same kind of kernel as other machines. In the case of MOSIX, this means that each machine must run a highly specialized Linux kernel. This may not be ideal for some users. Challenges #3, #4, and #6 are left unsatisfied.

2.5 Cloud computing

Experiences with institutional computing services have often served as motivation for researchers to investigate cloud computing systems. A number of systems have been built to allow for easy deployment of virtual machines [35–37] in commercial hosting services. These systems allow their users to specially tailor services to grant challenges #1 & #2 at the expense of challenges #3 & #4.

A hybrid approach of sharing computational resources among consumers and providers would be ideal if there were only some way to address challenges #5 & #6. The usual solution to this problem is to use VMs to provide not only dynamic introduction and migration of resources, but also insulate resource providers from malicious users. This still places a burden on the end-user to recruit sympathetic computer owners as well as maintain the security and integrity of a computing environment. All the hypothetical student really wanted to do was run an application.

A number of recent systems such as [23, 38–41] are evolving to address *some* of these challenges. They attempt to support applications that are generally CPU-bound, parallelizable and *interactive*. However, each of these systems requires additional burdens on the application developer or end-user such as kernel modifications on the host machine, additional hardware support, administrative privileges for execution, recompilation of the program, or special engineering of the application.

For example, Xax [39] requires program recompilation or linking against special libraries. This is a difficult task for ordinary computer users and also frequently impossible with commercial, closed source software. Moreover, Xax does not support process migration. UCOP [38] suffers from similar limitations while trying to address many of the same goals as MX. Other systems such as Jettison and Xen-Blanket [40, 41] offer migration while operating on entire VMs, requiring the end-user to maintain and secure an entire OS image. Moreover, the granularity in this case limits one's ability to properly load balance.

2.6 The role of machine virtualization

In recent years, the ready availability of various forms of machine virtualization has made it possible for persons not skilled in the art of maintaining heavy machinery to build their own virtual clusters and connect them with virtual networks. Companies now exist from which one can rent such virtual resources for a period of time. Because of their economy of scale, these services are likely to gradually replace smaller localized institutional computer support centers. Nevertheless, the same questions of maintenance and administration of the machines as well as how to set up and distribute applications in the environment still apply.

Machine virtualization also allows individuals to use their personal computers as hosts for virtual cluster nodes. By doing this an end-user could more easily contribute incremental resources to a homogeneous computing environment regardless of the type of physical machine or host operating system. For instance, someone with many complicated computational jobs could create an operating system image to run in a typical desktop virtual machine and recruit people to run it. Although this is easy enough for volunteers, this creates a burden and liability for the job distributor to keep those remotely executing operating systems up to date and secure.

The observation we make is that virtualization is an enabling technology that gives end-users the flexibility to use their own resources for many uses such as running multiple OSs and types of applications or ensuring secure isolation of untrusted applications. Virtualization also enables large organizations to commoditize the resources for end-users who wish to create generic computing environments without having to worry about physical infrastructure and maintenance. Nevertheless, virtualization has not significantly enabled advances in distributed computing because end users do not have the means to apply it to an execution venue where they lack administrative control. Moreover, platform virtualization does not assist in issues of selectively exposing end-users' local environments to the remote computing domain.

Our primary influence for userland kernel implementation is User-Mode Linux [42, 43]. UML is a port of the Linux kernel that runs on another Linux kernel for the purpose of creating a virtual machine. An unprivileged user of a Linux host can invoke a UML kernel with a block-based local filesystem and the resulting virtual machine can support virtual processes. Because UML uses the infrastructure of the Linux kernel, it is very similar to a classical monolithic kernel on a conventional machine. It has no support for distributed operation other than virtual network facilities that allow applications to connect to network services.

3. MOTIVATION

Analysis of 2008 data from nanoHUB.org, a cyber-platform supporting the National Nanotechnology Initiative (NNI) [44], reveals 394,000 simulations of which approximately 17% (68,000) were sourced to computing resources including the TeraGrid and OSG as well as local clusters. These applications consumed 1,800 CPU days while the users experienced 8,000 days wall clock time waiting for results. 6,200 days were lost waiting in job queues.

A more detailed analysis was conducted for three specific applications—nanowire, nanowireMG, and CNTfet. The problems that these applications address are domain-specific and of little interest aside from the fact that they require significant computational resources to execute. There were approximately 54,000 simulation jobs created for these three applications over the course of the year. These jobs were generated by 5,000 experiments that were run by some 1,000 users. Each job required on average about one hour of CPU time to run to completion. Each job waited, on average, four hours to start. This type of scenario provides the primary motivation for our work. We wish to allow ordinary end-users to execute scientific applications on conventional computing resources without being subjected to the wait time associated with a queue, even if this requires overcommitting resources.

We also want to support applications that effectively act as interactive calculators, allowing a user to dynamically provide parameters in the form of files as well as free-form input. A trivial example would be that of a numerical solver system such as Octave [45]. Here, overcommitment is necessary since a user may invoke it, interactively prepare for a calculation, and then use a great deal of CPU time once that calculation is started. One could argue that this program might be more suited to run on a personal computer rather than in a distributed environment, however more substantial examples of this type of interactive application include those that involve

large-scale parallel execution and only run on machinery that is presently beyond the reach of casual users.

With overcommitment comes an increased need for load balancing to avoid situations where, for example, a single machine becomes heavily loaded while others are functionally idle. Neither the load characteristics nor the duration of an interactive application can be known *a priori*, so distributing load only at the start of each application will not be sufficient to balance load. Furthermore, an application that runs in a queue-based computing environment must finish in the time quantum that has been allocated for it regardless of how many extra overcommitted applications have slowed its progress. This not only precludes the introduction of interactive applications with indeterminate run duration, but also complicates the decisions as to which resources have enough time slack to support overcommitment. As a result, any solution to these limitations must be able to migrate running processes.

We see two primary modes of operation for such a system. The first is a native mode where the job execution middleware is invoked by a person who wishes to provide a personal resource such as a desktop machine or local cluster to other end-users. In this case, the provider may wish to regain control of machine resources and require eviction of any running processes on short notice. The second mode of operation is as an overlay system where the middleware, itself, is invoked as a regular job in a conventional HPC environment. In either case, instances of the middleware can support new jobs either by introduction or migration. In the event that a computing resource is subject to a queue timeout, existing jobs can be migrated elsewhere as long as at least one middleware instance is running at all times.

Since the goal of our system is to reduce the human costs of distributing computation, we stipulate that end-users should be able to introduce unmodified native applications to the job management middleware. That is, no one should have to recompile or otherwise modify an application to distribute it. If an application works in the local environment where it was developed, it should behave the same way when run through the middleware without modification. In the same way, administrators

of computational systems should also not be burdened with changing the resources to support the middleware. Instead, the job submission middleware should run as a normal application that supports running, migrating and checkpointing other virtualized applications.

Today, large-scale computing is done almost exclusively with clusters of individual systems. Each of these systems provides support for a basic application environment and a high-speed network interface to its neighbors. An end-user is expected to obtain access to a gateway and use a conventional job distribution system [5,16,17] that can support multiple instances of a scalar or network-parallel application [21,22,24]. Such a state of affairs is unfortunate for two reasons. First, the user must construct an application specifically for machines that support the execution environment rather than for the machine on his or her desktop. If the user wants to use data that exists on a personal machine, that data must first be moved to and later retrieved from the execution environment. Second, the execution environment cannot make use of the potentially idle resources of the user's personal machine unless it is, as we suggested previously, incorporated into the execution environment as a first-class member.

This situation provides the motivation for our work. We wish to allow a naïve end-user to take the environment available on his or her personal machine and project it into an execution domain without invoking any of the sticky questions of software homogeneity, trust, ownership and management. Symmetrically, we would like such a user to be able to offer his or her computer resource for use by others in the larger distributed environment—again, without having to force a change in selected platform. To meet this goal, we set out to develop a distributed kernel that runs as a native, unprivileged application on a conventional OS and supports dynamic introduction of user clients as well as computational server resources.

4. SYSTEM OVERVIEW

To understand the structure of our distributed kernel, it is best to start with an overview of how the system is used. Consider the following generic application, `prog`, that accepts arguments, file names, and uses input and output redirection:

```
$ prog -x -y /path/to/a/file > out < in
```

All one must do to run this application in a way that enables migratory execution is:

```
$ mx prog -x -y /path/to/a/file > out < in
```

Simply prepending the `mx` command causes the specified program to be executed by a *client* in the distributed environment. The user has immediate and transparent access to all of his or her local files and is able to leverage the processing and memory resources provided by the server(s).

Another way to run the same program would be to first start an interactive shell and use it to invoke the application.

```
$ mx /bin/sh
$ prog -x -y /path/to/a/file > out < in
$ ls -l out
-rw-rw-r-- 1 usr1 grp1 15053 Oct 13 16:33 out
$ wc -l out
503 out
```

This starts an interactive shell inside the distributed system at which point any additional commands or programs are automatically executed inside of the system by

virtue of the fact that the shell itself is virtualized. Full support for interactive TTYs allows users to pretend they are still working on their own personal computers while leveraging an arbitrary number of other systems to execute their programs—all with transparent access to client resources.

In the event that users wish to start their own servers, the invocation is equally simple:

```
$ mx -s
```

To simplify invocation, client and server instances of `mx` refer to a configuration file that specifies the initial network topology (connections are authenticated and optionally encrypted), access rights for the client VFS, network access policies, user credentials, and various run-time optimization parameters. Construction of the configuration file is generally a one-time operation and does not get in the way of typical use. More detailed examples of invocation and use are discussed in Chapter 9.

Our approach to creating a distributed system revolves around a userland, distributed kernel. This kernel virtualizes execution of native Linux applications in private *containers* by abstracting the OS ABI through use of the ptrace [46] interface. The general mode of operation for our system involves multiple physical machines, each running at least one invocation of our kernel. We refer to a kernel that maintains containers as a *supervisor*. An end user can invoke a kernel as a *client* to launch an application into the system.

Each instance of the kernel communicates with peer instances using a socket. A client can connect to the distributed system and initiate a *session*, forming an overlay network that spans one or more kernels. Collectively, these kernel instances maintain a number of coherent, distributed data structures for each session. These structures are used to effect synchronization and distribute/share data throughout the session. In addition to acting as a client or supervisor, any kernel can be configured to act as a *relay* that provides additional routing and aggregation of other kernels.

4.1 Sessions

A new client joining the distributed environment forms its own session that can span multiple servers, each of which is assumed to be running on a different physical machine. Figure 4.1 shows an example of this. Each server can support multiple sessions, all of which are independent from each other. Each client and server in a session is represented as a logical *endpoint* which communicates with other endpoints using a flexible virtual routing scheme. If two clients' sessions share a server on a physical machine each session will still have its own independent endpoint, process, filesystem and network namespaces. This is the case with Servers Y and Z in the figure. Client A creates a session that spans three servers that host endpoints numbered 1, 2 and 3. Client B creates a session that spans only two with endpoints 1 and 2. This illustrates how clients can share multiple servers without contention. Hereafter, we will only discuss endpoints within a single session.

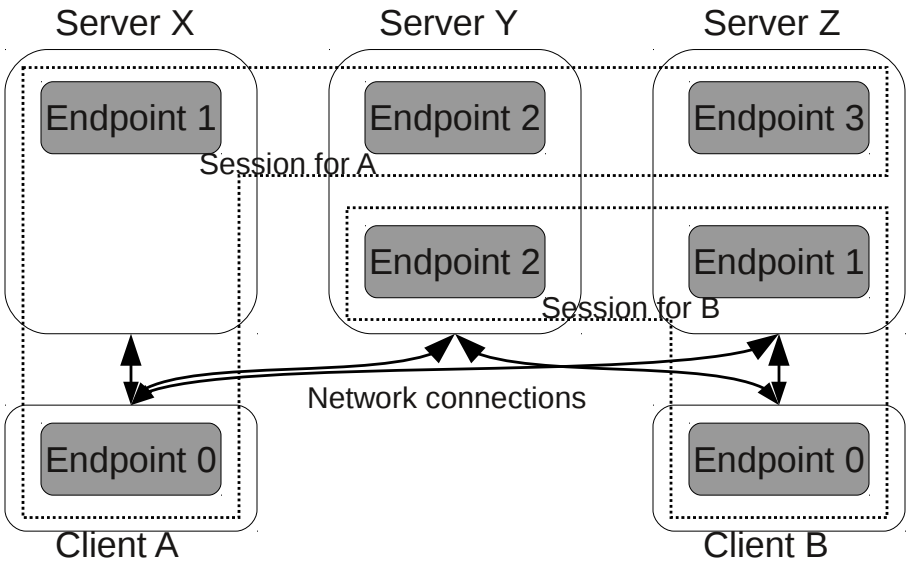

Fig. 4.1. Distributed system session example

Each client submits to its session an application that consists of a native executable. Each server presents to its session a kernel application binary interface

(ABI) that allows a native executable to appear to be running on a normal system. Each server can support multiple processes for each endpoint in each session that it participates in. To support encapsulation of potentially malicious applications each process runs in a container and interrupts the server only when it faults on memory or invokes any system call. The mechanism used to implement such containers is similar to that used by Consh [47] and User-Mode Linux [42, 43]. For our development we chose a subset of the Linux x86 system calls (both 32- and 64-bit flavors) as the ABI presented by the servers. This is not the only ABI possible or desirable, but it is representative of the domain of applications we want to support. In time, we intend to allow servers to support multiple ABIs simultaneously.

When a client connects and creates a session, endpoints are created that serve as routing mechanisms and sites of execution or data sharing (e.g., a file server). All messages (aside from session initiation traffic) are sent within a session to endpoints. This completely abstracts the underlying network, eliminating any knowledge of physical IP addresses and network topology that would interfere with transparent migration.

When a client executes a program in our system, it appears to be running locally on the client's workstation. The important difference is that it is able to leverage the processors and memory of potentially thousands of remote servers.

Our system is similar to Amoeba [26], MOSIX [27], Sprite [28, 29], and V [30] in that it presents the illusion of a single machine to the end-user. There are, however, several important differences that set our system apart from historical distributed kernel implementations. First, rather than running an installed operating system on a hardware platform, a user invokes our system by invoking a program. This program runs with the user's standard privileges and requires no modification of the host operating system or file system. The program is responsible for presenting selected portions of the user's environment such as the local file system, applications, and command arguments to the distributed kernel. It can be thought of as a *client* in a traditional client-server system. This client invokes a *session* in the distributed

kernel and provides information to that session about what application to run as well as details about the user's environment.

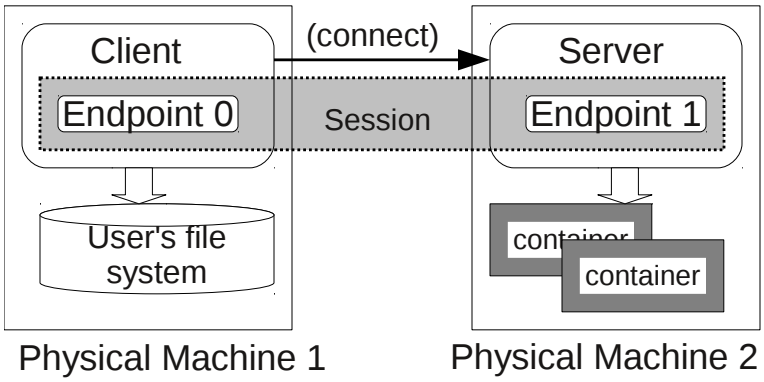

Fig. 4.2. Example of a simple distributed system session

Second, servers in our distributed kernel are also regular programs. These servers, when running applications on behalf of clients, supervise the execution of *containers*—processes that are totally under the control of the server. The simplest environment that we can create with our system consists of a single server and a single client as shown in Figure 4.2. For example, the user can create a server on an arbitrary machine and on another machine start a client that connects to that server using a bidirectional socket. All communication between the client and server takes place on this socket. The server never needs to create another connection back to the client. The server also does not need to share a file system with the client. In fact, it is not even necessary for the person starting the server to have the same identity as the person who starts the client. The server has an authentication system that is separate from the host operating system. In addition to platform independence, it allows someone to start a server that other persons can use. When establishing a session the client and server each create an *endpoint* that serves as a source or a destination in the session's routing operations.

Third, the server can be configured to support more than one session from more than one client. For instance, after starting a single server on a machine an end-user

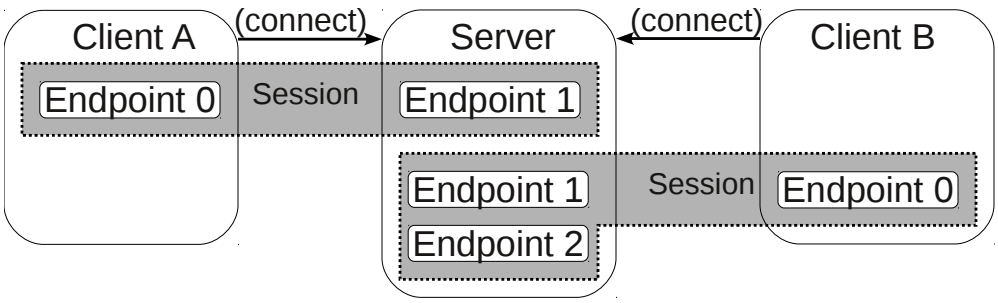

Fig. 4.3. Invoking two clients creates two sessions

might want to start an application with one client and start another application with another client before the first has finished (Figure 4.3). Each client starts a new session and the server handles each endpoint of each session independently. Although the two clients share resources they are isolated from each other and effectively see two separate virtual machines.

In some circumstances a client may request multiple endpoints as is the case with Client B in Figure 4.3. Usually, multiple endpoints would be placed on different physical machines. Since there is only one server in this example, there is no other place for Client B to add the other endpoint.

Finally, each distributed kernel element can also be configured to act as a relay between clients and servers. Such a configuration can have arbitrary topology as shown in Figure 4.4. Clients do not need to know if a system is acting as a server or relay (or both). The session initiation mechanics find the least loaded servers with the shortest path to a client. This allows the provider of the computational resources to provide connection redundancy, bandwidth balancing, the ability to aggregate multiple servers behind a single relay, and to hide details of the networking system. For instance, in Figure 4.4 Servers 1, 2 and 3 might be on a private network that is only accessible to Relays 1 and 2. If the servers connect to those relays, Clients A and B need only connect to either relay to be able to reach the servers. This diagram illustrates several other intricacies: Servers 1 and 4 connect to only one

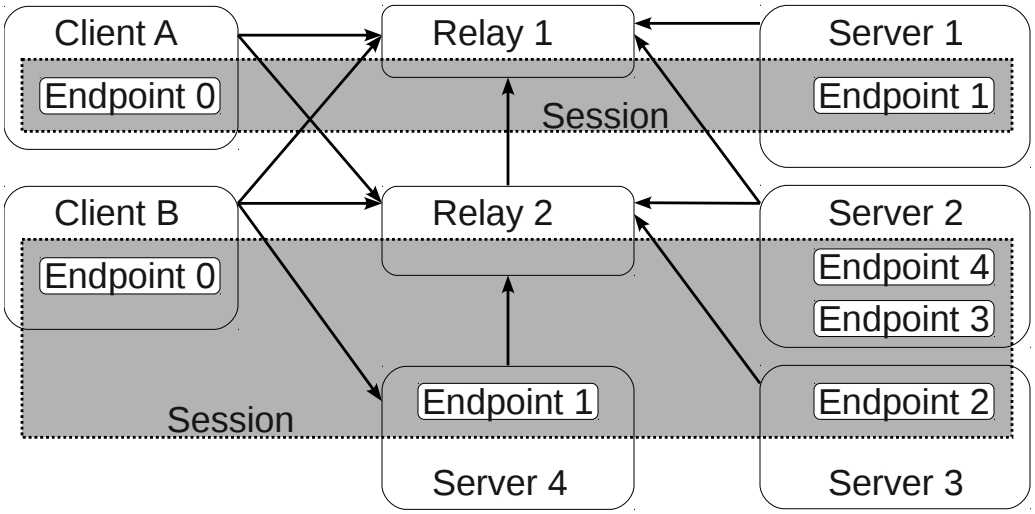

Fig. 4.4. A larger distributed session topology

relay—potentially because of misconfiguration or some legitimate policy. Server 4 may be reachable by everyone through Relay 2, but Client B is authorized to connect to it directly. Although several routing loops exist (such as the connection from Relay 2 to Relay 1), the resource discovery mechanism is not affected by this and will generally chose the shortest path between endpoints. The topology of endpoints that comprise a session is entirely independent from that of the network.

4.2 Transport

Endpoints in a session communicate by sending variable-sized packets of information to each other through a virtual overlay network. Each endpoint has a numeric address and the session information on every element of the distributed kernel includes a routing table. At the lowest level, standard TCP/IP sockets are used to transport packets between physical machines, but endpoints are unaware of the IP addresses at which they reside. This is done primarily so that all resources managed within any endpoint can migrate to some other endpoint in a session. It also ensures that clients and servers are anonymous to one another for reasons of security.

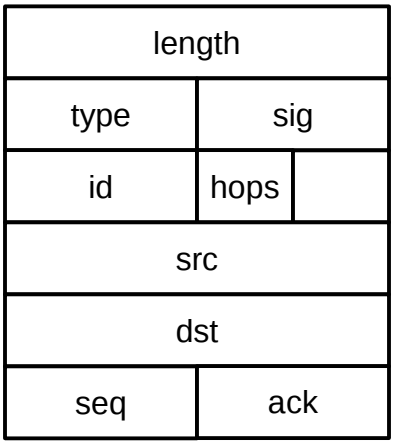

Fig. 4.5. Message header

Each packet sent through the system includes fields to indicate what kind of message it represents as well as a sequence number to guarantee proper ordering of messages sent from one endpoint to another. This is important for implementation of distributed coherence protocols. The payload of each packet is encoded using a message system that handles platform-specific details such as byte ordering. The header for such messages is illustrated in Figure 4.5.

- *len*: length of payload in bytes
- *type*: message type (e.g. `VNODEREMAP`. First 8 bits are object type (e.g. `VNODE`, `PPAGE`, `DISCO`, etc). Second 8 bits are subtype.
- *sig*: signature of encoded data
- *id*: entry to use in receiver's *gatemap*
- *hops*: number of network links traversed so far
- *src,dst*: source, destination endpoints
- *seq*: sending endpoint's outgoing message count to destination

- *ack*: sending endpoint's received message count from destination

The signature is an aggregate field that is updated each time data is added to the message. It is similar in many respects to CRC polynomials. Its primary purpose, however, is not error correction as much as it is a development-time verification to ensure that a message has been written and read in the correct order.

When a message arrives for an endpoint it is received by the server via a socket connection. Each socket is associated with a *port*. The server looks up the endpoint number, or *ID*, in the *gatemap* found on the port. Each element in the gatemap points to a *gateway*. Gateways map remote endpoint IDs to local IDs. They also provide a pointer to the appropriate session. Each session in turn has its own *route* structure that points to the actual endpoint, if present on the node, and to a routing table that knows to where to send a message to reach a certain endpoint. Figure 4.6 illustrates these relationships. For a given endpoint there is also a *sequencer* that maintains sequence numbers and acknowledgments, guaranteeing proper message ordering and facilitating checkpointing.

While almost all message handling conforms to the above description, there are a small number of "session-initiation" messages that cannot be addressed to endpoints. These messages are used to bootstrap endpoints into existence. Thus, session creation involves a handshake negotiation that results in the registration of remote and local IDs and the creation of a gateway and its associated reference in the gatemap(s).

4.3 Distributed coherence

A primary mechanism that lies at the foundation of our distributed kernel is a distributed coherence implementation we call "DisCo." This implementation is based on the fixed distributed manager algorithm proposed by Li and Hudak for Ivy [48]. We have extended this system by adding support for manager changes and asynchronous releases. Our implementation is also designed to support generic data types instead

of fixed pages. This allows us to enforce a coherence protocol on any type of object in our system.

Distributed coherence comes with a cost [49,50], of course, and is used as sparingly as possible in our system. In particular, the kernel runs in its own private address space on each machine. That is to say, there is no attempt made to make the real processes that *implement* our distributed kernel coherent. By contrast, the Ivy system was built in such a way that the kernel as well as the applications ran in the context of the distributed shared memory. In our kernel, coherence is applied only when needed on the representation of objects in the distributed system like virtual file system objects, virtual process memory regions, and a few other necessary objects. In depth discussion of our approach to distributed coherence takes place in Chapter 5.

4.4 Virtual file system

Our system, like other distributed kernels such as Sprite and LOCUS [29, 51], offers its own network file system. This file system projects a view of the client's file system into the virtual environment with appropriate access policies as defined by the end-user. When a file is opened by a process, a vnode is created with maps to regions in the physical space. These regions are populated on demand with the file's contents by the client and are kept coherent as described in Chapter 6.

In regular operation, this file system also acts as a unification file system [52] in that changes made in the distributed system are not reflected on the client side until termination. This serves as not only a convenience to cache data in the virtual system where it is being used but also as a security mechanism to prevent a malicious server from compromising items in the client's local file system.

Our file system differs from other network file systems in two important ways. First, the view presented to all running processes inside the system is sequentially consistent and fully coherent—even for a single file. We want to make the distributed system look as though it is a single machine. Within a conventional operating system,

file coherence between processes is usually taken for granted. When these processes are spread out on disparate endpoints on distant machines it is necessary to build the mechanisms to support that illusion as efficiently as possible.

Second, named FIFOs, TTYs, and sockets are supported in the file system as first-class elements. Each session represented by the distributed kernel has only one endpoint where a user would interact with a device file such as a TTY. As processes throughout the system interact with a TTY the distributed kernel arbitrates their accesses so that writes do not overlap or get lost and lines of input are sent to all processes that are waiting on them. Processes must be able to communicate through named FIFOs and sockets regardless of which endpoints they reside on. By comparison, typical network file systems such as NFS [14] do not support the ability for processes on different machines to communicate through device files, named FIFOs and sockets in the network file system. Such files can only be used for communication between processes that reside on the same physical machine. The VFS is discussed more in Chapter 6.

4.5 Processes

A single-threaded process is represented in our distributed kernel by a *task* data structure and a memory vnode. A multi-threaded process consists of multiple tasks that share the same memory vnode. Each task contains the mostly-immutable state of a process such as:

- the *personality* of the task (This is the ABI that the kernel must provide for the task, such as Linux-x86_64. This field is changed only when an `exec()` system call is executed.)
- the process ID, parent process ID, etc. (The parent process will change only when the parent exits.)
- the user and group affiliation

In our distributed kernel, there is no need for a task to change its user credentials. Every process in the session represents only one user and this user is the only principal operating on the local physical file system of the client. There is no need to represent multiple users through mechanisms such as `setuid()`. In any case, since the end-user invokes the session through an unprivileged client program there is no way for it to change its credentials to *act* as a different user on his or her local file system.

The task structure also contains mutable state such as:

- information about the container in which this process executes
- the file descriptor table that references all open files
- the resource usage and limits
- a list of child processes
- the vnode for the current working directory

Most other mutable state is held within the container used to execute the process and is only occasionally pulled into the server when needed. For instance, at various times such as during system calls, memory faults or migration the task data structure temporarily stores the contents of the container's registers and a copy of the current system call arguments.

As the container for the task executes the application requested by the client it interprets events consisting of memory access faults or system calls in the container that are intercepted by the server. The validity of each access fault is first checked against the memory vnode for the task before the memory is made available to the container. Each system call can query or modify the task state or initiate some kind of virtual file system operation through the client. We describe the implementation of process virtualization in detail in Section 7.

A task may `fork()` and create a new task. Either of these tasks may migrate to a new endpoint. When they do so, the vnodes they refer to as well as other transport information are shared.

4.5.1 System calls

Our system virtualizes *all* system calls to some degree. In more complicated cases where arguments include pointers to userland data, the supervisor first copies these arguments, inspects them, and then when necessary injects a new system call into the container pointing to arguments on the scratch page. This ensures that a process can not exploit time-of-check to time-of-use (TOCTTOU) races. A more in depth discussion may be found in Chapter 7.

4.6 Contexts

An interesting aspect of our system that manifests through iterative development is what we call a "context." Contexts are essentially simplified, migratable stacks that are used when a supervisor function must suspend or defer while waiting for a response from the client (or maybe another server). Similar in many respects to traditional stack ripping [53] our approach allows pointers and other opaque data references to be encoded and marshalled appropriately. By heavily relying on the C preprocessor we are able to abstract and simplify the application of this approach.

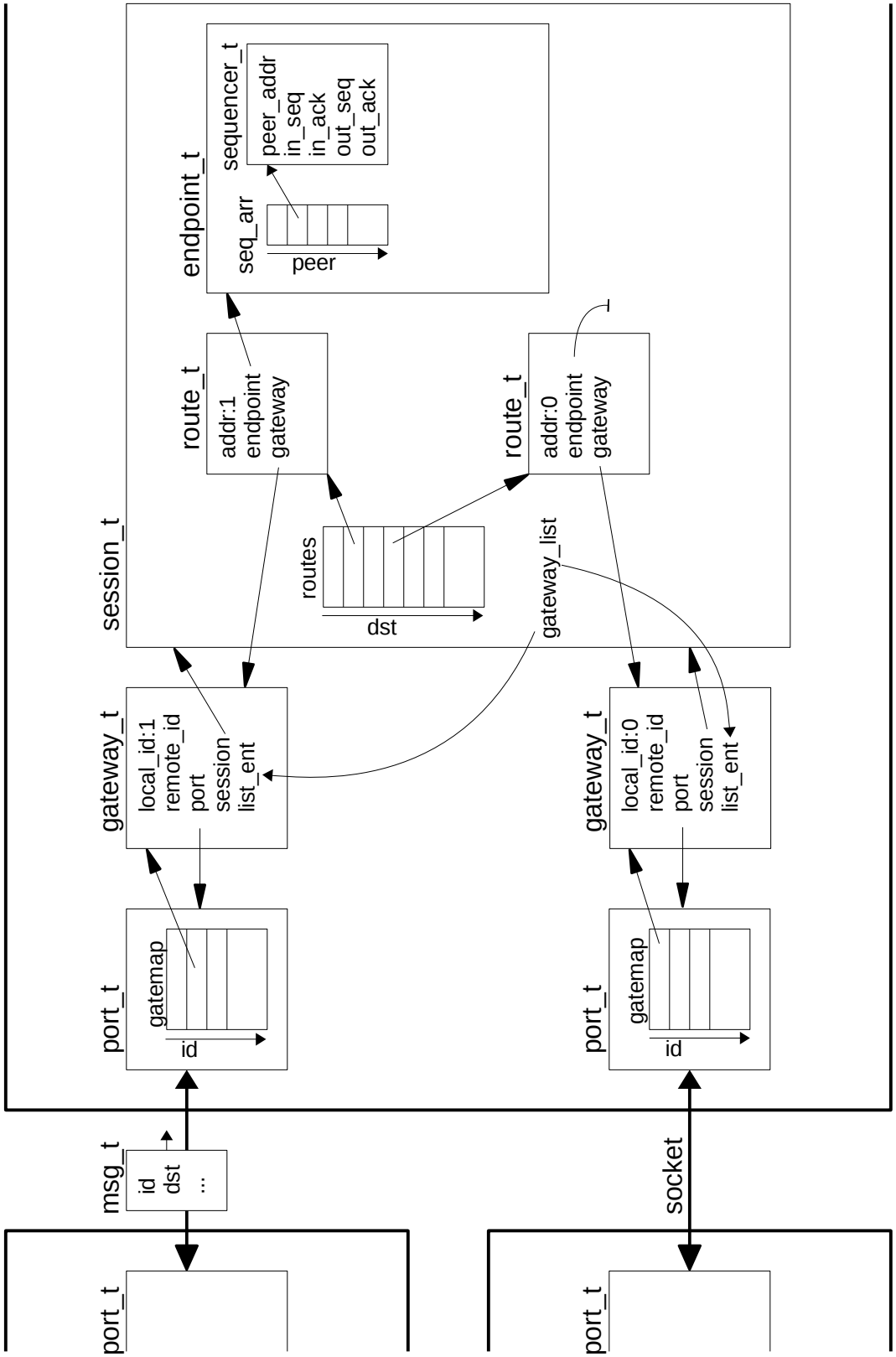

Fig. 4.6. Message transport layer

5. DISTRIBUTED COHERENCE

Our primary method for synchronization in this system is a distributed coherence (DisCo) mechanism. This mechanism implements the fixed distributed manager algorithm proposed by Li and Hudak [48] coupled with a modified MOESI coherence protocol [54] to allow for synchronization between endpoints. The fixed distributed manager has been modified to permit manager changes as well as asynchronous resource releasing, two things that were unnecessary for its use in Ivy. DisCo allows one to guarantee sequentially consistent access to protected objects, an important prerequisite for supporting many mechanisms in this distributed system.

For each object protected by DisCo in our distributed kernel there is the notion of a *manager* and an *owner*. The manager simply satisfies the role of a directory. All endpoints know the manager of all DisCo structures. The manager is the only endpoint that always knows which endpoint is the owner. The owner is responsible for delegating access control to different endpoints. When an endpoint wishes to modify a distributed resource, it becomes the owner. The owner is responsible for maintaining the *copyset* (list of sharers) as well as sending invalidations in the case where exclusive access for a write is needed. As long as no updates are being made to an object, many endpoints can share the object for read-only access.

It is worth mentioning that the fully-distributed manager algorithm proposed in [48] is inapplicable in this particular system due to the fact that the existence of an endpoint is not guaranteed and may be transient. This fact results in broken probOwner chains that cannot be easily reconciled. Existing work does focus on maintaining probOwner chains in such an environment, but at the cost of increased overhead. [55]. Moreover, the fact that the client often serves as the file server further limits the benefits realized from executing such an algorithm.

Our DisCo mechanism utilizes a structure composed of the following fields:

- *type*: what kind of object is being protected
- *lock*: synchronizes multiple accesses by different processes as well as remote requests
- *state*: corresponding MOESI state for the data
- *manager*: endpoint responsible for knowing the owner of the object
- *owner*: endpoint that is presently responsible for maintaining the *copyset*. This field is only valid on the manager
- *copyset*: contains the endpoint numbers that have read copies of the page. This field is only valid on the owner.
- *invalidating*: boolean value used by the owner to detect one of the race conditions discussed in Section 5.1.1

There also exists a queue that records endpoint numbers and intents for outstanding requests that encounter a locked DisCo. The above fields facilitate the construction of the following primitives that provide the basis for accessing and managing DisCo-protected objects.

- `init()`: allocates a DisCo structure, initializes the fields, and sets the type-specific ops function pointers
- `lock()`: tries to atomically obtain the lock on the DisCo structure
- `unlock()`: releases the held lock. Fails if lock is not held
- `queue()`: used when `lock()` fails to delay request until lock is available
- `invalidate()`: engages in requisite activity to guarantee exclusive access for the owner
- `fetch()`: request read or write access to a resource

These primitives in turn are used to construct an interface primarily composed of two actions,

- `access()`: the workhorse for DisCo objects, this function returns a boolean value indicating whether or not the desired access (`READ` or `WRITE`) is allowed at the current point in time. If it is not allowed, operations to obtain the appropriate access are initiated
- `release()`: returns a resource to its manager. If caller is the manager and at least one other endpoint holds the resource a manager change is initiated

The operation of the underlying primitives closely mirrors the operation of Ivy [48]. `access()` represents a read or write that in Ivy would potentially trigger the page fault handler. We implement a similar handler, but its use is not restricted to paging. There is no parallel for `release()`.

5.1 Differences from Ivy

While the bulk of DisCo mirrors Ivy, there are a number of key differences. The first and most obvious is the fact that the above coherence structure is used to protect generic data objects and not a fixed page of memory. It can be used to effect synchronization throughout an entire distributed system, instead of just distributed shared memory.

Being able to synchronize access to disparate object types is of particular interest to us because in consideration of performance the proposed distributed operating system kernel primarily runs in its own private address space on the local endpoint. Only userland data and necessary portions of the kernel are kept coherent. In other words, the kernel itself is not running on top of a distributed shared memory.

5.1.1 Resource releasing

The second difference, induced by the fact that endpoints are not static in the system, is that resources may be released. Ivy's implementation relies on a strict request-response messaging paradigm. Until a request is received, the resource (a page) lives in perpetuity on the last requester. Releases that happen without a request create unexpected complexities in the context of a coherent system.

A number of situations may arise when an asynchronous release is initiated that would not occur in the original Ivy implementation.

- The release can cross paths with an invalidation from the owner
- The release can happen while the DisCo object is locked
- The release may happen at the same time as an ownership change
- The release may happen at the same time as a manager change

When a release message crosses an invalidation message, it is simply dropped on the receiving end. The receiver knows that it is in the process of invalidating the resource on other endpoints and therefore the sender will be forced to implicitly release the resource anyway. Moreover, this situation cannot occur if it is the owner releasing the resource, as only the owner may initiate an invalidation operation.

If a release encounters a locked DisCo object, it is simply queued as any other request would be.

If an ownership change has occurred, the message is forwarded to the new owner if it is a sharer doing the release. It is possible that the owner is instead releasing and this release has crossed paths with a request from a new owner. Because ownership changes happen immediately on the manager, it will detect the release from the old owner, see that the originating endpoint is no longer the owner, and drop it. The request from the new owner will already be in flight to the old owner and will result in an implicit release of the resource.

5.1.2 Manager changes

Occasionally, a manager change may have to take place for certain DisCo-protected resources. This happens, for example, with vnodes when all tasks referencing the vnode on the managing endpoint either exit or migrate elsewhere. If one or more endpoints are still maintaining a copy of the vnode, one of them should become the new manager. In particular, this is necessary if the managing endpoint is to be destroyed. Thus, we extended the algorithm to support synchronized manager changes.

This problem was addressed, in a sense, in the Ivy paper by the dynamic distributed manager algorithm. Their solution is inapplicable, however, due to the reasons discussed earlier.

Manager changes are a real possibility in this system due merely to the fact that endpoints can be transient. For instance, suppose a task creates a pipe, `fork()`s, and uses that pipe to communicate with its child. Now, suppose both tasks migrate elsewhere. By virtue of the fact that the vnode was initially created on the original endpoint, that endpoint is the manager. However, once both tasks have migrated elsewhere, there is no reason for the original endpoint to maintain a copy of the vnode. Thus, a manager change is initiated.

Manager changes are possible because each DisCo structure maintains an *open_list* containing a full and accurate list of all endpoints that may manipulate the DisCo-protected data. It is also this scenario that reveals the existence and reasoning behind the *close_list*. If a resource is in the process of closing, it should not be selected as the new manager. However, until the final phase of deletion is reached the endpoint remains on the *open_list*. This is necessary so that coherence is maintained in the event of an aborted delete. A more detailed example of deletion is discussed in Section 6.2.4.

When a manager change is initiated, an endpoint from the *open_list* is chosen that is not also on the *close_list*. Moreover, if an endpoint that is not on the *close_list* is

the current owner, it is preferred over others. Once the new manager is chosen, the deleting endpoint immediately sets its manager field to the new manager. All DisCo messages, in turn, have a check that, if upon receipt of a message the receiver is neither the owner nor the manager, causes it to be forwarded to the manager. It is by this mechanism that DisCo messages "chase" the new manager, potentially through multiple manager changes.

The old manager then sends a `VNODECHMAN` message to all endpoints on the *open_list*—even to the new manager. The message sent to the new manager also includes the owner, remote_count, and remote reader and writer counts

Upon receipt of a `VNODECHMAN` an endpoint immediately updates its manager field for the resource and replies with an `ACK`. After this point, any DisCo message will be sent to the new manager.

When the old manager has received `ACK`s from all endpoints, this guarantees that it can no longer receive any DisCo requests (by virtue of the fact that message ordering between any two endpoints is guaranteed). At this point it may free the resources associated with the vnode.

5.2 Distributed mutual exclusion

Our system occasionally has implementation details that result in a manifestation of the classical critical section problem. These are code segments that require mutual exclusion to serialize access to a particular object or code region. It is for this reason that we introduce a distributed mutex.

Distributed mutexes have been discussed at length before, and there are a variety of (rather complicated) implementations that have been proposed [31, 56]. Building mutexes on top of a distributed coherence algorithm allows us to avoid a great deal of complexity experienced by other implementations.

The distributed mutex contains the following fields:

- *lock*: used for mutual exclusion and protection of address field

- *queue*: list of endpoints waiting on the lock

This mechanism leverages DisCo to guarantee coherent access to and manipulation of the mutex. Because the kernel on each endpoint is single-threaded, no additional kernel locking is required when manipulating the mutex. There are, however, some tricks that must be played to ensure message quiescing. A concrete example of something that uses such a primitive can be found in Section 6.2.3 where we discuss Vnode Remapping.

6. VIRTUAL FILE SYSTEM

In this chapter, we narrow our focus to the filesystem implementation that allows our distributed kernel to present a consistent data view to running processes while hiding those complexities. Synchronization is particularly important for this part of the system and the distributed coherence layer on which it is built was discussed in Chapter 5. The rest of this chapter is organized as follows. In Section 6.3, we outline a set of primitives and operations that allow a local file system to be projected into a distributed system while maintaining expected block-based file system semantics. Section 6.4 details the additional filesystem requirements needed to support interactive streams such as FIFOs and sockets.

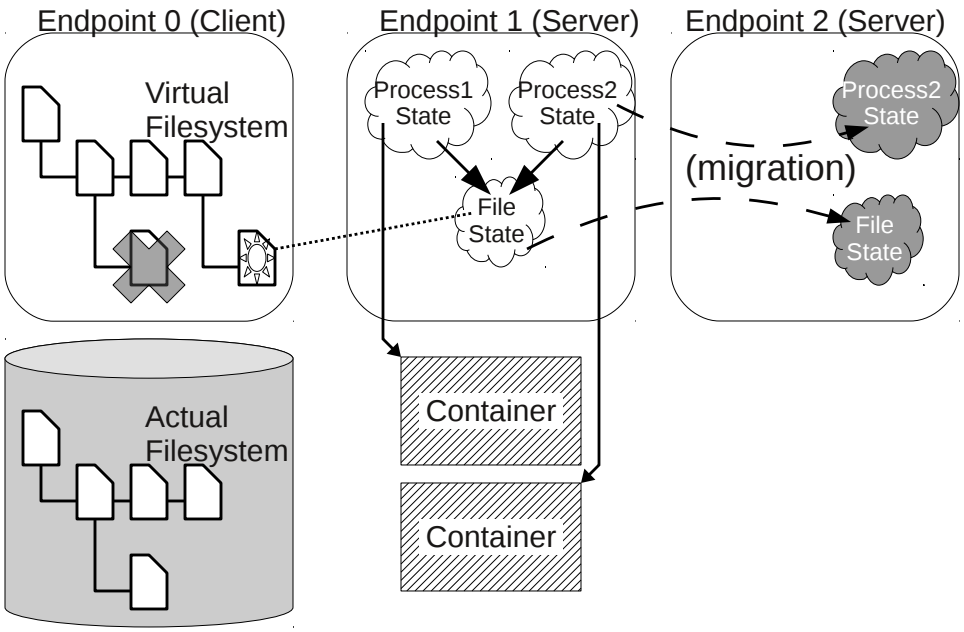

Fig. 6.1. Integration of distributed tasks with the filesystem

Figure 6.1 shows how the filesystem is integrated with process containers on each endpoint of a single session. The VFS pulls necessary files from the actual filesystem. Changes such as deleted files and newly created or modified files are held entirely within the VFS. This is effectively a unification filesystem [52] that permits read access to native files subject to policy rules and caches written or newly created files and directories in the VFS layer. The native platform's filesystem is insulated from executing programs until termination where the user can selectively choose which changes to commit and impose policies on what constitutes a legitimate operation.

There are two principle challenges involved with developing our distributed kernel. First, we must hide distribution complexities as well as latencies of the underlying implementation from the execution environment. Second, we must also allow more sophisticated operations such as checkpointing and process migration to occur transparently. For instance, if Process 2 shown in the figure migrates from Endpoint 1 to Endpoint 2 the migration must also move the representation of the newly created file that it shares with Process 1. It must do so in a fashion that preserves the access semantics expected as if the two processes were running on a conventional operating system.

6.1 Physical space

We introduce the notion of a "physical space," [1] contained in Figure 6.2. The physical space is essentially a traditional distributed shared memory implementation that allows for the coherent addressing and accessing of regions of data. These regions may vary in size, but are constrained to a power of 2 for simplicity. Each region is addressed by a 64-bit value. On startup each endpoint is allocated a 4GiB chunk of the physical space. When a chunk is allocated or reclaimed, all other endpoints are notified. An endpoint that is responsible for a given chunk is known as the manager. Importantly, this physical space is common across all endpoints. In other words,

[1] physical space is a misnomer—there is nothing physical about it, but it is analogous in many ways to physical memory in a conventional operating system

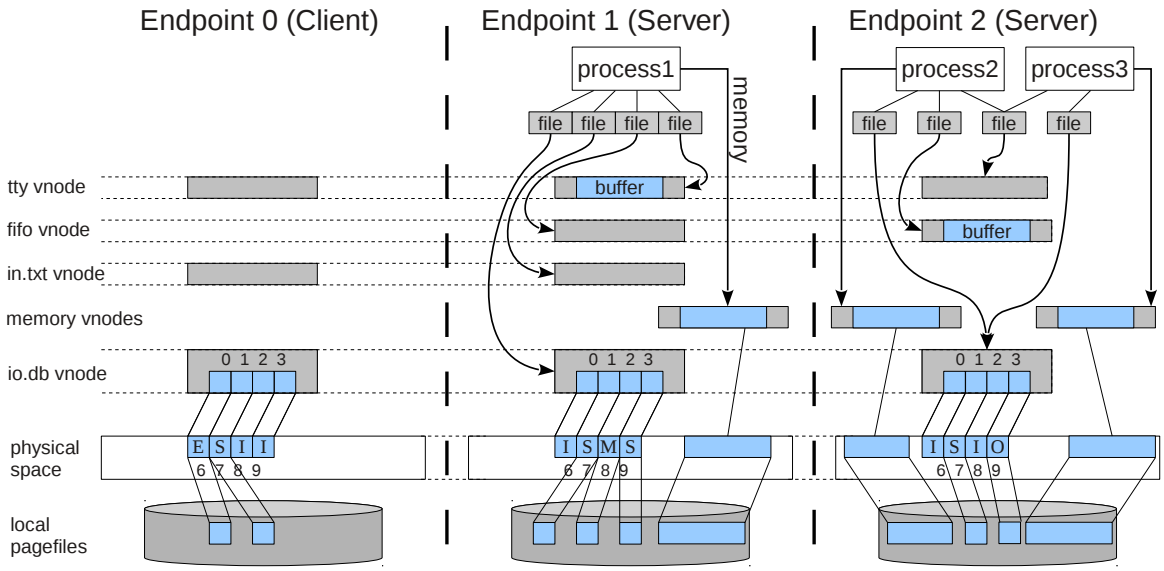

Fig. 6.2. Vnode and physical region coherence in our distributed kernel

every endpoint in the system knows the manager for a given range and may become the owner at any time.

For instance, four physical pages in Figure 6.2 are shown with their MOESI state. This physical space provides a cache for file system data as well as a location in which to store the memory of virtualized processes.

When a task accesses part of a file, the various levels of indirection are traversed until a "physical region" is obtained. This physical region contains a DisCo structure that guarantees coherent access to the underlying data via the `access()` primitive described earlier.

The physical space is analogous in almost every way to a traditional distributed shared memory and makes migration possible in our system. When a task moves to another endpoint, none of its references to locations in the physical space need to be updated. Moreover, if multiple tasks executing on different endpoints have the same file open, our system provides the same operating semantics as if they were both running on the same single machine.

It may seem odd that the overhead incurred by this approach is acceptable, but the common case—particularly with execution of scientific applications—is that memory and file data sharing is rare. When sharing happens, it is frequently read-only. Even in the case of read/write sharing, however, the performance penalties are acceptable as illustrated in Chapter 9.

6.2 Virtual inodes (vnodes)

In addition to the physical space above, our system implements a structure that is analogous to an inode in a conventional OS. This virtual inode, or vnode[2], contains coherent stat information as well as mappings to regions in the physical space. Like a conventional OS, these vnodes represent directories and files, as well as process memory. Because even directories are virtualized, we can provide a fully-distributed virtual file system. For example, when a process opens a file in our system, a server-side `namei()` occurs. Assuming the file exists, a coherent copy of its vnode is sent to the opening endpoint. This vnode provides mappings into the physical space, where the contents of the file are cached.

When a file is opened for the first time in the system, a request is sent to the client. The client then creates a vnode that contains virtual regions pointing to one or more regions in the physical space, also illustrated in Figure 6.2. There is a one-to-one relationship between physical regions and file blocks, but there may be a many-to-one relationship between virtual regions and physical regions. This type of relationship only occurs for process memory. In this system, memory is simply another vnode.

A file is not immediately copied into the physical space when it is opened. Instead, a region corresponding to its size is allocated in the physical space, mappings are set up in the vnode, and a client provides that data on demand to the system when a particular physical region is requested.

[2]not to be confused with Vnodes in Sun Unix [57] or BSD

When a vnode is used to represent a virtualized process's memory, resident data from the physical space is stored on the local endpoint's disk. The underlying physical machine's `mmap()` semantics and block caching ensure that memory access overhead is negligible. [58]

In order to manage access to a particular vnode from multiple endpoints in a sequentially consistent manner, both the physical space and portions of the vnode must be kept coherent. The more complex aspect in this system is vnode coherency. It is the vnodes that provide the mechanism for obtaining a particular block's physical address in the system. As such, it is possible for a particular vnode to exist on multiple endpoints. Moreover, vnodes come and go as tasks open and close files. This transience leads to a number of difficulties discussed further below.

Vnodes leverage the invalidation-based coherence protocol provided by the DisCo implementation as well as a separate update-based coherence protocol for mappings.

Figure 6.2 represents a session with a client endpoint and two server endpoints. This session has two open files represented by "vnode 1" and "vnode 2". Vnode 1 is only open by the task on endpoint 1. Vnode 2 is open by both tasks. For vnode 2 we show the mapping of its virtual regions into physical space. It is important to note that it is mapped into the same part of the physical space on each endpoint. Physical space provides the mechanism by which data are shared across all endpoints.

For a vnode that represents a stream such as a TTY or a FIFO, there exist additional coherent structures that track the active reader and active writer. Only one endpoint may be the active reader or writer at a given point in time. Changes to the active reader/writer must be synchronous and are therefore accomplished via DisCo. Any endpoint may attempt to become the active reader/writer at any time.

As input becomes available, it is proactively forwarded to the endpoint where the buffer resides, assuming sufficient space exists. This endpoint is also the active reader, and as long as that is true, any tasks on the endpoint may read from the buffer. When a process on another endpoint wants to read from the stream, the buffer must be stolen from the previous endpoint and moved to the requesting endpoint. The

active writer must also be aware of this change. Such changes are permitted to occur asynchronously. Should the former endpoint holding the buffer receive data from the active writer it simply forwards it to the *new* active reader.

The stream implementation greatly favors the situation where there is a single reader and writer for a stream at any time. The mechanisms for changing the active reader and active writer are only invoked when necessary. This is generally restricted to migration and, in rare cases, where multiple readers or writers exist on disparate endpoints. We find that it is overwhelmingly the case that for a given stream there exists only one reader and one writer. In this case, only migration would cause invocation of expensive semantics.

6.2.1 Vnode layout

The basic structure of a vnode is as follows

- *stat*: traditional filesystem stat information
- *vregion_root*: pointer to the root node of the red-black tree used to maintain all virtual regions
- *open_list*: endpoints that have this vnode open
- *remote_count*: number of (remote) endpoints with this vnode open
- *write_through*: should changes to the underlying file appear immediately, or at program termination?
- *valid_ranges*: byte ranges corresponding to written data used for write-only files

The virtual regions maintained on the vnode are kept coherent through an update-based protocol that does not guarantee updates are seen at the same time by all endpoints. Traditional operating systems lack this guarantee as well due to the asynchrony introduced by the fact that only a subset of processes are running at a given

point in time. The managing endpoint must know to which other endpoints it must send updates, and this provides the motivation for the *open_list*.

The *remote_count* field is simply used to detect a termination condition. When this count drops to 0 the client knows that it can release its handle on the underlying file system. When all vnode counts drop to 0 the client knows that there are no running tasks and the session should be terminated.

When an underlying file can only be accessed for write, traditional coherence still applies for the underlying physical pages. However, only regions that have been written to are actually valid when it comes time to reconcile the overlay with the client file system. For this reason, a coherent list of valid regions is maintained on the vnode. This list enumerates the regions of an vnode that have been written to and are, therefore, valid.

It is imperative for stat information to be distributed, due in no small part to the findings of AFS that many applications incessantly and frequently stat the same file for often no apparent reason. [59]

The use of an update-based protocol for vnode mapping coherence is possible in part due to the fact that mappings for files are never removed. They are only added via an extend operation, discussed in section 6.2.2. Even in the case of truncation, the mappings remain present. The vnode stat size is simply updated to reflect the fact that the data located through those mappings are no longer valid.

Of final note is the fact that a vnode can operate in two modes—write through and write back. Write through mode is used when one wants changes made by a virtualized task to show immediately on disk (i.e., immediate data access by a non-virtualized process is needed). The other mode caches all changes until termination, giving rise to an overlay file system that merges with the client file system on exit.

6.2.2 Vnode extend

Occasionally processes will write beyond the end of a file (particularly if it has just created the file). In this case, additional mappings must be added to the vnode and new pages in the physical space allocated. Such an operation is another example of the critical section problem. Only one endpoint may manipulate vnode mappings at a time or risk corruption and race conditions.

To resolve this problem, vnodes utilize the distributed mutual exclusion outlined in section 5.2. Before any endpoint adds mappings, it must hold this mutex. With the distributed mutex, mapping modifications become a routine problem. The following pseudo code describes what happens when additional data must be added to the end of a file.

```
disco_access(WRITE, extend_mutex)
if (!lock(extend_mutex.lock) {
  queue(extend_mutex.queue, me)
} else {
  // Allocate new physical page(s)
  // Create new mappings
  // Send updates to everyone on map_list
  // Wait for ACKs
  unlock(extend_mutex.lock)
}
return
```

It is important to note that extend operations only *add* mappings to the vnode. They do not change or remove existing mappings. Only after additional mappings are added is the vnode's stat size updated to reflect the increase in file size. As a result, other endpoints may continue to reference *existing* mappings while an extend operation is in progress. They may not, however, start their own extend operation. Such code is guarded by the mutex.

6.2.3 Vnode remap

As implied in the previous section, when an endpoint must add data to a vnode, it allocates physical page(s) in the region of physical space for which it is the manager and adds mappings to these pages to the vnode. Because the physical space is coherent, any other endpoint may dereference these mappings and, through DisCo, may request read or write access to the pages at a later point in time.

This works well for regular operation, but in the case of endpoint destruction, something must be done with the physical pages managed by the terminating endpoint. Such a problem exists for process memory as well, in the case of task migration. If there is no dependency on a vnode for a given endpoint, the vnode should disappear and any physical pages allocated on that endpoint should be remapped to a region of the physical space managed by a remaining endpoint that still has the vnode open. If no such endpoint exists, then the vnode should be closed as described in the next section.

A vnode remapping operation is more involved than the extend operation discussed above. In particular, we are now involved in modifying *existing* mappings so other endpoints cannot continue to reference the vnode maps while a remap operation is in progress. As such, the mutex for remapping guards *any* access to the vnode mappings, whereas the extend mutex only prevents concurrent extend operations.

This layered approach to solving the problem makes implementation and description rather straightforward while abstracting a great deal of complexity.

6.2.4 Vnode closure and release

When a vnode is closed on a remote endpoint, there is a three-phase deletion process that is begun. A number of things must happen in sequence to ensure that closure happens properly and that synchronization issues do not arise between other endpoints. The following lists are used to ensure proper synchronization:

- *open_list*: endpoints with this vnode open

- *close_list*: endpoints that cannot be valid targets for a chman operation

These two lists are managed as one object, supported by one DisCo.

When all tasks on a particular endpoint that reference a file have terminated, there is no longer any need for the vnode to continue to exist on that endpoint. It becomes necessary to "release" the vnode. Phase one involves the following:

- Obtain write access to the protected lists on the vnode.
- Add self to *close_list*, preventing use as a target for a manager change.
- If this endpoint is the manager and other endpoints have the vnode open, initiate manager change operation, and wait for it to complete.

While the manager change operation is in progress, it's possible for a task on the releasing endpoint to open a file that references the vnode. If this happens, the chman operation is allowed to complete but deletion is aborted. The vnode is still in a usable state, although it is now managed by another endpoint. Once a vnode moves beyond this point, deletion cannot be aborted, as the vnode is no longer coherent with other copies. The following operations are then performed:

- Re-obtain write access to DisCo, if necessary.
- Remove self from *open_list* and *close_list*.

It is safe to remove the endpoint from the *close_list* at this point because the above mentioned scenario is no longer possible. The client uses the *open_list* to determine whether or not an endpoint already has a vnode and the deleting endpoint has removed itself from the *open_list*.

- Release all dirty physical regions referenced by the vnode.

This is necessary to ensure that any modifications to a write-back file are preserved. Two different things happen at this point. Either the releasing endpoint is

the manager (and therefore the only endpoint with the vnode still open because the *open_list* is empty), or a different endpoint is the manager. In the former case, the endpoint simply frees its local resources, completing the deletion. In the latter, the following phase two happens:

- Release the vnode DisCo resource (wait for ACK).
- Release the valid_ranges DisCo resource (wait for ACK) if `O_WRONLY`.
- Transmit an `VNODECLOSE` message.
- Free local resources.

When the manager receives an `VNODECLOSE` message it implicitly traverses all physical regions referenced by the vnode removing the sending endpoint from any copysets and, if appropriate, changing the state. This reduces message overhead considerably, requiring the closing endpoint to only return physical regions that have been modified.

This process is broken into two phases because each phase may restart multiple times while waiting on an action. If combined, phase one obtains the vnode DisCo resource, and phase two releases it. This would result in a rather boring infinite loop of sorts as releasing the DisCo inevitably causes a phase to defer and be re-invoked upon release completion.

6.3 Block-based file access

Whereas a conventional OS typically has a single inode per file and one block pool per file system (Figure 6.3), our distributed system has multiple block pools as well as multiple copies of a single vnode. This is illustrated in Figure 6.2. Not only must the storage blocks be coherent, but portions of the vnode as well. It is also important for all endpoints to be advised when the size of a vnode changes. In the case of an increase in size, mappings to new blocks may also need to be added. Probably

the least expected complication requiring intimate involvement of coherence is vnode closure or release. The following sections describe how all of this is accomplished.

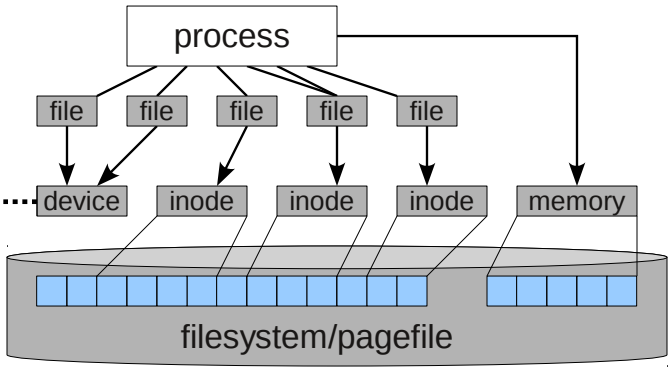

Fig. 6.3. Data structure layout in a conventional OS kernel

6.3.1 File descriptors vs. vnodes

It is important to make a distinction between a file descriptor and an vnode. Multiple file descriptors may reference the same vnode. A file open that results in a first-time access of an vnode causes the creation of the vnode and a client-side *open()* using the file descriptor's flags (e.g., `O_RDONLY`, `O_WRONLY`, `O_RDWR`, etc). As subsequent files are opened that reference the same vnode, the underlying file on the client is reopened to contain a superset of all file modes when possible. If such a reopen isn't possible, the appropriate error code is returned. For example, the underlying file may be marked write-only.

6.3.2 Write-back operation

For regular read-only I/O our approach is simple and fast. Blocks are obtained once and remain on as many remote endpoints as necessary. Writes to existing regions

are also simple and invoke the regular write-invalidate operation that exists in any MOESI protocol.

Even when a file is only available write-only the system behaves normally due to the sequential consistency guarantees provided by the underlying DSM. Extend operations take place as needed to obtain new mappings, and writes proceed as usual. The key difference is that reads are prohibited, and written regions are tracked by a *valid_ranges* field, which is kept coherent by its own separate DisCo structure. On termination, these ranges are written back out to the underlying file on the client

Even when one encounters files that are open for disparate modes on different endpoints—e.g., if two endpoints have a file *open()* write only and a third is reading from it—the system behaves normally. In this case the vnode itself is open read/write. It is the file descriptor flags on each endpoint that determine userland restrictions.

6.3.3 Write-through operation

Write-back operation corresponds to a fully virtualized overlay file system and, as such, is relatively straightforward. Write-through operation, however, comes with its own set of complexities.

In write-through mode the end user wishes to see internal progress externally. Presumably this user is willing to pay a penalty for this type of access. As a result, whenever a write to a file-backed vnode occurs, in addition to the regular semantics described above involving the physical space, the writing endpoint additionally sends a `VNODEWRITE` message to the client. This message causes a direct write to the underlying file, assuming the user's established policy allows it.

It is worth noting that files that have been `mmap()`'d have unpredictable behavior, just like in a conventional operating system. If the client happens to obtain a dirty copy of one of the physical regions, it will write it back. This happens lazily, however. This is the case in, for example, Linux [60].

6.4 Streams

Stream I/O includes pipes/FIFOs, sockets, and character devices (ttys). In each case one has the traditional problem of multiple readers and writers accessing a shared resource. The primary difference between streams and block-based vnodes, however, is the lack of blocks and therefore the underlying physical space discussed previously. This prohibits leveraging the aforementioned DSM for consistency.

6.4.1 Stream protocol

We developed a stream protocol implementation that facilitates communication between a single reader and writer. This protocol alone, however, is insufficient to ensure proper operation when readers and writers are on different endpoints. Nevertheless, this is an important part of the puzzle.

The stream communication protocol mirrors many standard implementations of any buffered communication protocol (e.g., TCP). Stream I/O is presently implemented with what can be functionally thought of as two "shared memory segments" each consisting of:

- *buffer*: circular buffer containing stream data
- *size*: maximum amount of data buffer can hold
- *start*: offset into buffer where data begins
- *amt*: amount of valid data in the buffer
- *seq*: the sequence number of the first byte in the buffer (like TCP)
- *remwinsz*: amount of space left in peer's buffer

One segment is used to maintain the active reader while the other maintains the active writer. Only one endpoint may be the active writer of a memory segment at one time. A DisCo-protected field, *active_addr*, indicates which endpoint is presently

active as the reader or writer. An endpoint is currently the active user of a buffer when:

- the DisCo is readable
- *active_addr* == the endpoint address

Stream buffers can be filled and drained by the active endpoint. Filling the stream buffer will deposit data in *buffer* and advance *amt*. Draining the stream buffer will extract data from *buffer*, advance *start*, and increment *seq*.

A sequence number must be provided for each fill. If the sequence is not equal to `((seq + amt) % size)`, it means that some data has been received out of order. A pending region is created for the missing region between `((seq + amt) % size)` and the start of the new data. While there are pending regions, a stream buffer can still be filled. A stream buffer cannot, however, drain a pending region.

In the case where all readers and writers for a particular vnode reside on the same endpoint this protocol operates just as a "normal" stream implementation would on a bus-based system. Writers simply deliver data into the circular buffer, blocking (or returning in the case of `O_NONBLOCK`) when it is full. Readers simply retrieve data from the buffer.

However, even when dealing with a single reader and writer it is possible that one exists on a different endpoint. In that case flow control is required to avoid overflowing the buffer. This leads to three messages.

- `STREAMPUSH`: deliver a payload to write into the buffer
- `STREAMSTAT`: a delta indicating any change in response to a `STREAMPUSH`

Transport consists of draining data from a writer stream buffer and either pushing it to a remote reader stream buffer or depositing it in a local reader stream buffer.

The writing endpoint maintains a "window size" which encodes the minimum amount of data that the buffer is capable of holding at any point in time. This

represents the maximum amount of data that can be transmitted and fill the buffer at one time. Data are sent in a `STREAMPUSH` message. When possible, window size updates are embedded in preexisting messages to reduce latency.

The reader simply obtains data at-will from the buffer and when appropriate notifies the writer via `STREAMSTAT` of any changes in the window size due to available buffer size changes.

This works fine in the simple case where all readers exist on a single endpoint and all writers exist on another (or the same) endpoint. Let us now consider the case where there are still two endpoints but a reader exists on both.

Because the buffer can only be present on one endpoint at a time we must now introduce two more messages.

- `STREAMREQBUF`: obtain the buffer
- `STREAMPUSHBUF`: deliver the buffer

These messages allow the stream buffers mentioned above to move between endpoints. Whenever such movement occurs, the *active_addr* field is appropriately updated.

6.4.2 Termination

Up until this point we have assumed that reader and writer existence has been static, or at least increasing. At some point, however, all readers and all writers will disappear (assuming the program involved terminates).

When there are no writers semantics dictate that *read()* calls should return 0. Moreover, when there are no readers *write()* should return `EPIPE`. Our description up to this point includes no mechanism capable of enacting these requirements. We therefore introduce the following fields:

- {*reader,writer*}*_count*: number of local readers and writers

- *remote_{readers,writers}*: dual meaning on manager indicates number of endpoints with active readers or writers, on all others boolean indicating presence of at least one reader or writer

The protocol for managing these four fields involves two messages which handle four distinct cases

- `STREAMPRESENT`: Increase the remote_{reader,writer} count by one
- `STREAMABSENT`: Decrease the remote_{reader,writer} count by one

There are two roles in this protocol - one for the manager and one for everyone else. When a non-manager's reader or writer count goes nonzero, it sends a `STREAMPRESENT` message to the manager. Correspondingly if either drops to zero, it sends a `STREAMABSENT`.

The manager, on the other hand, sends a `STREAMPRESENT` message to all other endpoints when the local or remote count goes nonzero for the first time and a `STREAMABSENT` when both return to 0.

Thus the absence of a reader or writer in the system is detected by all endpoints when both the local and remote count is 0.

There is no synchrony between endpoints determining that all readers or writers have left the system. It is unneeded because this type of synchrony is lacking in a traditional operating system. At any point in time, a task may have been preempted and therefore no longer be running. Only when execution resumes and the task invokes an appropriate system call is it notified that there are no longer any readers or writers.

6.4.3 The common case

The above handling of multiple readers and/or writers is not inexpensive. There is no getting around that fact. However, it is important to realize that the common case overwhelmingly involves a single reader and writer. It is rare for a program to

leverage streams to implement IPC between more than two processes. We view this as an acceptable trade-off.

6.4.4 Closure and release

As discussed in Section 6.2.4, vnode closure is complex even in the case of block-based vnodes. Streams complicate it further and require the introduction of a third phase between phases one and two described in Section 3.6.1. After phase one is complete, if the endpoint is either the active reader and/or the active writer it must also release the resources associated with each by:

- Selecting a new endpoint to be the active reader or writer
- Pushing the buffer to the new reader or writer

Because stream buffers contain a DisCo structure, the DisCo itself must also be released. This happens in phase three for both the active writer DisCo and the active reader DisCo. This is once again to avoid the scenario where in the act of detecting whether the endpoint is the active reader or writer it obtains read access to the DisCo and subsequently releases that access in an infinite loop.

The same abort semantics exist up until phase three for streams. At any point prior to removal from the *open_list* deletion may be aborted. In that case even though the endpoint may no longer be the active reader or writer, it is in a coherent state that would permit it to re-obtain such status. Phase three is thus:

- Release the vnode DisCo resource (wait for ACK)
- Release the active reader DisCo resource (wait for ACK)
- Release the active writer DisCo resource (wait for ACK)
- Transmit a `VNODECLOSE` message
- Free local resources

At this point the vnode has been successfully deleted.

7. PROCESS VIRTUALIZATION

Efficient remote execution of processes requires a virtualization mechanism with as little intervention as possible. Since we are not encapsulating a conventional kernel that expects to have access to hardware primitives such as TLBs and page tables, we do not need to support full machine virtualization. We also do not want to pay the performance costs of instruction-level virtualization or translation. Instead we want to let an application run natively and trap only exceptional behavior—namely page faults and system calls. This implicitly means that an application can only be run on the type of microprocessor for which it was written. For instance, a program containing i686 instructions cannot be run on an ARM CPU. However, the OS application binary interface (ABI) need not be the same between the virtualized application and the host. As long as the host can trap the exceptional events and it can emulate the operation of the expected OS ABI, we can fool the process into believing that it is running on a native platform.

In many cases one can accomplish this virtualization without the use of any privileged mechanisms. The technique we detail in this section is based on Linux `ptrace()`. This interface is the basis for program debugging as well as for other virtualization systems such as User Mode Linux [42, 43]. Other Unix-like operating systems have similar debugging mechanisms that can be substituted just as easily [47].

We describe the use of `ptrace()` to implement virtualization of native Linux executables (of either the i686 or x86_64 ABI) running on Linux. At present, this system implements a subset of the available system calls, but is already mature enough to run a representative set of applications. We are also writing support for more hosts and ABIs, but these are not ready for evaluation.

7.1 Container bootstrapping

When a session in our distributed kernel is initiated, the client delivers information about an application to a server. The server, acting in its capacity as a *supervisor*, creates a container process to support the application's execution. To do this, the server `fork()`s and `exec()`s an ordinary executable that does the following operations:

1. It `mmap()`s a single page of memory, at the highest location in the process's address space, to an external *pagefile*; copies few bytes of assembly code to it; and sets its protection to read-and-execute-only. We call this the syscall page since it will be used by the virtualization system for system call injection.

2. It sets the stack and data size limits to zero. These limits do not take effect until their memory pages are unmapped.

3. It sets its registers as they would be at the entry point of an `munmap()` syscall that clears the entire address space from zero up to the syscall page.

4. It jumps to the syscall page to invoke a system call.

Once the syscall is invoked, the container has no writable memory pages. It can only read from the syscall page. Execution continues in the syscall page to do two more operations: It closes the file descriptor of a pipe to notify the supervisor that it has succeeded in clearing its memory. Finally, it executes a system call that will sleep forever (usually `select(0,0,0,0,0)`).

When the supervisor receives the pipe close notification that the container has cleared its memory, it invokes the `ptrace()` mechanism to take control of the container. This interrupts the container's `select()` system call and allows the supervisor to redirect execution elsewhere.

7.2 Server-side delegated `execve()`

A new process introduced into the distributed kernel is created to look as though it had been created on the native platform through a call to `execve()`. All `execve()` operations are completely implemented by the container's supervisor on the server. To perform an `execve()` the supervisor goes through the same steps that a conventional operating system would:

1. It opens the vnode for the executable, reads the first block, and parses its headers to find the addresses of the program segments.

2. It creates mappings in the task's memory vnode for the text and data segments into the correct region of the executable.

3. In the event that the executable is dynamically linked, it repeats the previous two steps for the dynamic loader (ld.so).

4. It sets up the initial stack contents from the program invocation arguments supplied by the client, the environment variables from the client, and an auxiliary vector that represents the target program environment.

5. It creates a mapping in the task's memory vnode for the stack and writes the initial stack contents into the pagefile.

6. It initializes the program's registers, sets the program counter to the beginning of the executable, and sets the stack pointer to the bottom of the stack.

7. It uses `ptrace()` to resume execution of the container.

It is important to note that none of the `execve()` operations have caused any change to the container. All of these modifications have taken place within virtual objects such as the memory vnode, the distributed physical space, and the task state. The container still lacks any pages of memory other than the syscall page. The supervisor has set the container's program counter to a location where there is no

page and instructed it to continue. The first thing that the container will do is page fault. This is guaranteed since the bootstrap procedure ensured that the stack and data sizes were limited to zero. There can be no automatic allocation of any memory regions of the container.

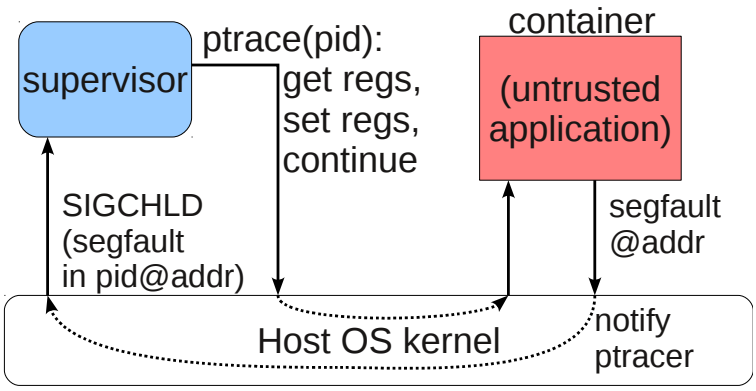

Fig. 7.1. `ptrace()` interception of a page fault

7.3 Page fault handling

When a process traced by `ptrace()` attempts to access unmapped memory, it notifies its tracer about the *segmentation violation* it has encountered. This notification includes information about the type of access as well as the address where it happened. This allows the container's supervisor to take actions to satisfy the page fault. To fix the problem, the supervisor first consults the task's memory vnode to determine whether or not the access is legitimate. If it is, the supervisor determines the location of the page in the pagefile and injects a system call into the container to `mmap()` it into the appropriate location. When this operation is complete, the supervisor restores the container's registers to the values at the time of the fault and tells the container to resume. This time, the container will be able to access the memory and continue on to the next exceptional event—either another page fault or a system call. Figure 7.1 shows the ptrace() relationship between the supervisor

and the container. When the container faults, the host kernel delivers the SIGSEGV information to the `ptrace()` supervisor rather than the container. The supervisor is woken with a SIGCHLD and reads the actual signal data about the container from the host kernel. At that point, it can use the `ptrace()` syscall to get or set the container's registers and continue its execution.

7.4 System call interposition

When a system call is invoked in the container it is trapped as an exceptional event by `ptrace()`. Once the supervisor is notified, it examines the container's registers to determine the syscall and its arguments, and decides how to handle it. For instance, if the system call is `getuid()` the supervisor will look up the virtual user ID it originally obtained from the client, *nullify* the container's system call so that it has no effect on the container when it executes, and place the result (the uid) into the result register before letting the container resume. Handling of system calls with more complicated arguments is explained in Section 7.5 in the discussion of the pagefile.

7.4.1 System call nullification

Once a `ptrace()`d process traps on a syscall, its execution cannot be diverted to another address. It must execute the syscall, however, the *kind* of syscall can be changed before execution. To nullify the syscall, we change the call number to something that can execute without having an effect on the process. The conventional choice is `getpid()`. Once the `getpid()` syscall has executed the supervisor will be notified of its completion. Before continuing the container again, the supervisor can change the register values to make it look as though the original requested syscall had its intended effect.

The supervisor intercepts and nullifies *every* system call. This is equivalent to the delegated architecture approach for secure system call interposition described by Garfinkel et. al. for Ostia [61]. While Ostia and others [62] rely on external

toolkits or kernel modules to ensure proper operation, we were able to implement all virtualization mechanics strictly in userland. This means that every system call is handled by our kernel, and also sheds light on why we call it a kernel. We are not simply engaged in system call filtering [63, 64] or system call wrapping [65, 66]. We are, in fact, offering a full fledged distributed kernel. All handling of the OS ABI is done by this kernel. When local client resources are required requests are always sent across the network.

Other remote I/O solutions like Parrot [67] simply redirect I/O system calls to a client process. This is insufficient for our goals due to a number of reasons. First, this approach does not support `mmap()`ed files. Second, inter-process communication becomes difficult. Third, this approach does not lend itself to advanced functionality such as migration and checkpointing.

7.5 The pagefile

Every page of memory used by a virtual process is stored in the pagefile. This is a dynamically-sized file that is shared between the server process and all of its container processes. From the time it is started, a container process has its descriptor 0 open to the pagefile. This allows the supervisor to tell the container to `mmap()` a region of that file into its memory. Every time a container writes to any region of memory, those changes are immediately available at a known location in the pagefile. Symmetrically, any writes by the supervisor to that location are immediately reflected in the address space of that container. This relationship allows the supervisor to make large-scale modifications to the container's memory without having to go through a cumbersome `ptrace()` interface.

It should be noted that while the pagefile is an actual file located on the host machine's file system, host block caching and `mmap()` semantics result in cached copies of these blocks residing in system memory. In most cases, pagefile accesses are simply memory accesses with delayed, asynchronous write-back to the host's physical disk

when the host's virtual memory is in short supply. Reads and writes to the pagefile do not incur the overhead associated with normal disk I/O [58].

An example of the use of the pagefile is handling of the `open()` system call. When the syscall is trapped, the supervisor obtains the arguments of the syscall. The first argument represents the address of the string containing the file system path to open. There is some chance that the region of memory holding this string may not be mapped in the container. Therefore, it would not be possible to use `ptrace()` to peek into the container to obtain the string. Instead, the supervisor looks up the virtual address in the memory vnode of the corresponding task, maps the virtual address into the physical layer, and finally maps the physical layer to the location in the pagefile. The supervisor can simply read the string from the location in the pagefile.

Another example of the pagefile use is when the supervisor encounters a `read()` system call. This syscall has three arguments that represent the file descriptor to read, the address in the process' memory where the results should be placed, and the maximum length to read. In servicing this syscall, the supervisor first checks the task's memory vnode to determine if the address and length represent a valid location. Upon going through the file system mechanics to obtain the data, the supervisor maps the container's virtual address to the corresponding location in the pagefile and writes the results there.

7.6 Deferred operations

Some types of operations involve many levels of action to satisfy. For instance, in some cases the page of memory that holds the first argument of an `open()` syscall may not exist in the container nor even be in the pagefile. This could happen if it were in a part of the executable that had not yet been fetched from the client. Since the distributed kernel uses demand-based paging for all operations such occasions are frequent.

In these cases, the supervisor will make the appropriate request through the network to obtain the necessary resource before resuming the container. Figure 7.2 shows an example where a `read()` syscall causes multiple network transactions before the container is permitted to continue. In this example, neither the page backing the `read()` destination address (`addr`) nor the block from the file it is reading from are available in the server endpoint's physical space. Executing the read operation requires two network transactions—one to fetch the page that backs the destination address and one to fetch the block of the file. Once both blocks are present, the supervisor can satisfy the `read()` by copying a portion of the file block into the target memory page. Whenever possible, we parallelize the execution of network requests as well as prefetch [68] to minimize latency. Assuming that a subsequent `read()` uses the same pages for the file block and destination memory address, no network operations will occur since the pages are already resident.

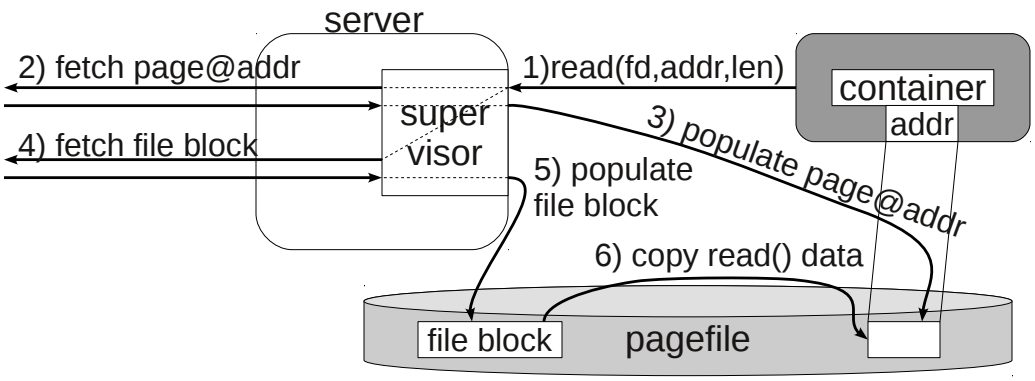

Fig. 7.2. Complex interaction between container and the distributed file system to service a `read()` syscall

7.7 System call injection

Although all syscalls are intercepted and nullified, there are certain operations done to service the requests by the virtual process that can only be done by the

container itself. For instance, when a page fault occurs, the supervisor lacks a direct means of creating a mapping from address region in the container to the pagefile. In such cases, the supervisor must force the container to start executing at the syscall page, set up the registers to designate the type of syscall to perform and then force the container to continue. The three major injected operations are `mmap()` to map new memory into the container as the result of a page fault, `munmap()`—to remove a region of memory from the container, and `mprotect()` to change the access protections on a region of memory. The majority of other operations can be performed by manipulating the memory of the container through the pagefile.

As an example of syscall injection, consider the case in Figure 7.1 where the container faults on memory location `addr`. Assume that the page containing this address is missing. The supervisor is notified of the fault. It checks the memory vnode to determine if the fault is legitimate, looks up the physical region, and looks up the block in the pagefile. It then redirects execution to the syscall page, replaces the syscall with `mmap()` to map the container memory to the pagefile block.

7.8 Event multiplexing

Our distributed kernel supports two approaches to handling its event loop—`pselect()` and `epoll()`. The `pselect()` implementation supports I/O across a maximum of 1024 network connections. It is interrupted whenever a supervisor must service a container event. The `epoll` interface is a new Linux subsystem that has no limit on the number of network connections (other than the process file descriptor limit). It also provides higher performance due to not requiring the four bitfield copies intrinsic to the operation of `pselect()` each time it is invoked. Unfortunately it is not supported on older Linux systems. The `epoll()` multiplexer also uses `signalfd()` to allow a server to wait on a signal from any container it supervises using only an additional file descriptor. This avoids the fundamental problems with race conditions between signal delivery and I/O polling.

7.9 Security considerations

Large scale distributed systems are often able to ignore most security implications due simply to the fact that access to the system is restricted to trusted users. The system proposed in this paper, however, is architected to permit arbitrary users to join an existing distributed system. This creates a question of trust that must be addressed for two cases: malicious programs running inside the system and malicious servers.

The structure of our system ensures that if a client introduces a malicious program it can only do harm to the client's system. This is guaranteed for a number of reasons. First, it is impossible for a virtualized process to escape its container by virtue of the fact that all interactions with the host operating system are nullified and instead handled by the userland kernel. Even if a malicious program were able to break out of its container, it would still only be able to assume the privileges of its unprivileged supervisor, essentially falling into the malicious server scenario below.

Because the kernel is a regular userland program, we are able to prevent time-of-check to time-of-use (TOCTTOU) races [63] by making copies of all system call arguments prior to their check and use. No address space is shared between the container and our kernel. This alone goes a long way to eliminate potential vulnerabilities.

The syscall page is also mapped read-only in the container, preventing any possibility of malicious applications writing to that location. Even if such a write were possible, breaking out of the container would still require the use of a syscall. Execution of syscalls on the syscall page still incurs oversight of the supervisor.

Although every container shares an open file descriptor to the same pagefile, each container can only modify parts of the pagefile that are mapped into its address space. For instance, the container cannot simply `read()` descriptor 0 to examine the pagefile. Such a syscall would be intercepted by the supervisor and translated into

an operation to read the vnode in the distributed file system that is referred to by the virtual task's descriptor 0.

For the case of a malicious server, we can assume that someone has altered a server to allow the modification of the virtualized program. Ultimately, this would allow the modifier a way to tamper with the contents of files on the client. We limit the scope of this damage by having the end-user explicitly enumerate the portions of the client file system that he or she wishes to export and modify. Moreover, due to the union file system-style overlay that our system implements, malicious writes are deferred until program termination where the end-user is presented a list of modified files and can selectively choose which ones to merge with the underlying file system.

Finally, we consider the case of a malicious server simply inspecting and modifying results. The general solution to this problem is to only allow clients to connect to resources that are trusted. This is no worse than the situation of a client using an institutional computing resource. A more thorough—albeit wasteful—solution would be to employ redundant execution [25] across multiple servers to reduce the likelihood that results have been corrupted. Note that this solution provides no assurance that a virtualized program has not been *spied upon* by a malicious server. For some applications this is just as bad a problem. Our distributed kernel currently has no provision for redundant execution.

8. MIGRATION

The architecture of our entire system is built to facilitate process migration and, at some future point, checkpointing. Process migration is made reasonably straightforward by the distributed coherence provided both for the physical space as well as other objects (e.g., vnodes). When a process initially migrates, minimal information is required to be transmitted to the destination endpoint. In particular, the following items must be sent:

- the task state—including registers, file descriptors, etc
- the memory vnode
- all vnodes corresponding to open files (Some may already exist on the destination endpoint.)

While vnodes are sent over to the destination endpoint, the underlying physical pages are not. Referenced physical pages are sent as needed when the process resumes. It is also possible that a preexisting process on the destination endpoint already has the vnode open and a number of the pages exist already. In that case no additional overhead is incurred.

8.1 Migration state machine

A state machine manages migration for a given task. This state machine is broken into the following main states:

- TASK: Marshal and transmit the task data.
- VNODES: Transmit a list of vnode numbers referenced by open file descriptors.

- TASKVNODES: Transmit a list of vnode numbers referenced by the task.
- MEMORY: Remap all non-file-backed physical pages.
- FDS: Transmit the file descriptor table, associated file structures, and referenced vnode numbers.
- DESTROY: Remove the old task and associated resources.

As described in Section 6.2, vnodes shared across endpoints are kept coherent. Vnodes are identified by a globally unique *number*, so when a task migrates we need only send a list of vnode numbers and the address of their corresponding managers. Upon receipt of this list the destination endpoint requests copies of each vnode from its corresponding manager (assuming that it does not already have a copy).

Once all vnodes have been received, an acknowledgment is transmitted to the originating endpoint which advances the state machine to the next state. The TASKVNODE state is similar to the previous one in that it again delivers a list of vnode numbers and their managers. This is short and contains only the memory vnode number and other miscellaneous vnodes that are intrinsic to the task.

8.2 Memory vnode

The memory vnode constitutes a special case for which a number of solutions exist. Our present approach is expensive but supports rapid evacuation of an endpoint from a server. One scenario for this would be when the owner of the computational resource demands that all guest computation be evicted so that he or she can enjoy full use of the particular machine. It requires the following messages:

- `PUSH_ANON_REMAP`: Deliver the contents of a physical page and require the receiving endpoint to place this payload in a new page for which it is the manager. Additionally map the included virtual page to this physical page.

- `ANON_REMAP`: This is an unwritten physical page. Do the same as above, but no payload is required.

- `ALIAS`: Map the specified virtual page to the physical page pointed to by the other included virtual page.

These messages are used to effect full remapping of the migrating task's memory into the portion of the physical space managed by the destination endpoint (for non-file-backed virtual regions). This ensures that physical regions corresponding to the task's memory are managed by the destination endpoint. Moreover, it frees up any dependencies on the originating endpoint, permitting it to be destroyed if needed. This remapping also eliminates the need for any future coherence actions as all pages will be automatically owned and writable by the destination.

This is essentially Charlotte's and LOCUS's approach to handling memory during migration [69, 70]. While it is rather expensive as it involves transmitting the entire memory for a process to a destination, there are a few things that are not transmitted. Virtual pages that reference memory mapped files actually point to regions in the physical space that are managed by the client. As such, these may be dynamically faulted in, and are not included in the bulk transfer above.

Unfortunately, if a process has a large memory map remapping can be quite expensive and result in a long "freeze time" [29]. An alternate approach that we have also implemented transfers the memory vnode to the destination endpoint but does not engage in remapping operations. This causes the migrating task to obtain all non-file-backed physical pages on demand from the originating endpoint as they are faulted on. The trade-off is that the originating endpoint must remain present in the system or invoke remap operations when it is reclaimed. This approach is similar to that taken by V and Accent [71–73], and is preferred for interactive processes as it minimizes freeze time.

8.3 Threads

There is little difference between thread migration and process migration. The main difference is that if at least one thread remains behind on the source endpoint, the memory vnode remains as described in the second approach to memory vnode migration above. A coherent copy of the memory vnode is simply transferred to the destination endpoint, and physical regions are faulted in on-demand.

8.4 Context switches and migration

When a subroutine in a conventional OS kernel needs to wait on a resource, it typically makes a context switch to a different runnable thread. Once the resource becomes available, the subroutine in the sleeping thread can resume where it left off with its entire original call stack intact. This works because the kernel can use a separate stack for each of its threads. Stack separation is not feasible for our distributed kernel, since we have no way to create a stack and ensure that it can be placed at the same virtual address in another supervisor when migration occurs. Furthermore, most of the local variables for a subroutine will be pointers to kernel structures that will have different addresses on the migration target. Although the subject of code mobility is well understood [74], solutions so far have been limited to fully interpreted code, extremely constrained control flow, or integer-only variables.

It would be impractical to simply wait for a resource to become ready so that the task could migrate when it is not servicing a system call. Some system calls wait for unbounded time (e.g. for `read()` or for any other call that can return EINTR). It would also be semantically incorrect to return EINTR to the calling process when it resumes. Migration must not be visible to the task, and most applications would be unprepared to handle EINTR without a prior signal handler invocation.

In order to preserve stack-oriented, yet still migratable, control flow within the supervisor, we introduce a *context* facility that can track the progress of a procedure as well as preserve its arguments and local variables in a *shadow stack*. Use of this

facility requires that each of the supervisor's procedures that is potentially subject to suspension receive a parent context as an argument, declare a context structure for itself that refers to its parent, wrap its local variables with a macro that reserves space for an encoded form in the shadow stack, and declare each of its significant stages by prefacing it with a *step label*.

When a procedure must wait on a resource, it invokes a sleep operation that walks through the context structure and its parents, saving the step label and local variables of each active subroutine into the shadow stack. For example, the encoded form of a pointer to a vnode structure consists of its virtual device/inode integers. A non-local goto (longjmp) then brings the supervisor back to a top-level handler without invoking any of the return code for the subroutines. When the resource becomes ready, the context list waiting on it is rebuilt from the top down using the step labels to re-invoke each intermediate subroutine where it left off. On re-invocation, the variable macros decode and restore the previous contents of variables rather than reserving more space. Since the type of each saved variable is also stored in the shadow stack, the supervisor can ensure that each item is available on a migration target and check the validity of the call stack as it is rebuilt. As an optimization, variables are only encoded if migration must happen. In the common case of re-invocation without migration, the shadow stack stores only raw pointers and avoids the encode/decode latency.

By using this context facility, the supervisor can trap a `read` system call, invoke arbitrary levels of subroutine calls to validate and handle the `read`, and sleep on the underlying vnode if it is not ready. The supervisor is then free to handle other requests. The supervisor context and shadow stack for this task can be migrated to another supervisor. When the vnode becomes ready to read, the original subroutine stack will be rebuilt and the call sequence will continue to completion. Return values from each subroutine will be used this time around.

9. EVALUATION

Virtual execution environments by their very nature are required to make sacrifices in performance to facilitate process, application, or even operating system isolation. It is obvious that, in terms of performance, a virtual environment can do no better than native execution. The addition of data transport over a distributed kernel poses a number of additional challenges, but at the same time offers a number of important opportunities. Parallelism available with multiple servers can allow us to compensate for the virtualization overhead with more widespread distribution of execution. Many of the latencies involved with servicing system calls and page faults necessarily invoke one or more network transactions to obtain the requested data. Compared to the network latency, the virtualization overhead is often inconsequential.

9.1 Virtualization overhead

To estimate the syscall handling time we wrote and compiled a program that repeatedly measured the time of each of 1000 invocations of the `getpid()` system call. To measure this time, we used the CPU hardware cycle counter before and after the syscall. We considered the minimum and average times over the ensemble. The minimum shows the lowest latency possible while the average represents the typical latency we expect to see in actual use. We first ran the program on a target machine and then ran a single server of the distributed kernel on the target machine and client delivering the test program to it remotely. The target machine was a 3.2GHz Intel Xeon W3570 running a Linux 3.0.0 kernel. The results are shown in Table 9.1. System call virtualization is typically 120 times slower—and no better than 76 times slower—than native execution. The latency is due to the supervisor having to repeat

the `ptrace()` notify-nullify-resume cycle twice. Each of the following steps of syscall nullification involves a context switch and unpredictable latency:

- The container notifies the host kernel that it has invoked a system call.
- The host kernel notifies the supervisor with a signal to indicate that the container has been stopped at entry to a syscall.
- The supervisor uses `ptrace()` to pull the registers from the container and decides how to service the system call.
- The supervisor nullifies the container's syscall and tells the host kernel to resume handling of the system call.
- The host kernel notifies the supervisor with a signal to indicate that the nullified syscall has finished.
- The supervisor uses `ptrace()` to pull the registers from the container and decide what to do.
- The supervisor writes the intended value of the syscall result register (and any other register values) into the container.
- The supervisor tells the container to resume.

Because of the frequent context switching, the absolute time of a *single* syscall is difficult to ascertain. When system calls happen infrequently in an application, we tend to see average latency. When they are tightly clustered together—as is the case with page faults—performance tends more toward the minimum case.

To test page fault time, we wrote and compiled a program that `mmap()`s a 16MB *anonymous* region—memory not backed by file storage. The first time any part of this region is inspected the value will be zero. Values written to this region persist for the duration of the program. The first access to each page will invoke fault handling. The test program was used in three different ways to step through the region of memory

Table 9.1
Overhead of virtualization of container exceptions

Type of container exception		Native (cycles)	Virtual (cycles)	Penalty
call `getpid()`	(min)	786.0	59442.0	75.6x
	(average)	1567.9	188051.6	119.9x
read fault		1329.5	90063.1	67.7x
write-after-read fault		3589.3	81826.3	22.8x
direct write double-fault		2924.4	170895.0	58.4x

with a 16KB stride to produce 1024 faults each pass. The test program uses the CPU's hardware cycle counter to measure the times just before and just after each page reference. We consider only the average times here. There are three different types of faults that are possible:

- A read fault happens when an absent page is read for the first time.
- A write-after-read fault happens when a present, previously read page is written for the first time.
- A write fault happens when an absent page is written.

The signal notification about a container fault tells the supervisor only about the current state of the page (present or absent) rather than the intent of the instruction that caused the page fault (whether it was reading from or writing to the memory location). If the page is present, the supervisor knows that the current permission of the page does not permit the container's access and it must resolve the fault by making the page writable. If the page is absent, the supervisor conservatively maps the page read-only—even if the mapping for the page permits writes. This is done so that an access to a shared virtual page does not result in having to obtain exclusive

access to the physical page. This also means that an initial write to an absent page will cause a double fault—first for the read fault and again for the write fault. In the results shows in Table 9.1 the "direct write double-fault" takes approximately the same time as a read fault plus a read-after-write fault.

Page fault handling is a heavier operation since it requires the supervisor to inject a syscall into the container and later restore the registers to make the container resume at the point where the fault occurred. Where a syscall would require the supervisor to force the container to continue twice, a fault results in it doing so three times. Table 9.1 shows that page fault handling is about 50% slower than the best case time for syscall handling. Write faults in a real operating system require resource management to handle situations such as copy-on-write. This is why the native Linux 3.0.0 kernel takes almost three times as long to handle a write fault (allocate writable backing-store) than it does to satisfy a read fault (reference an empty zero page). By comparison, our virtual kernel splits regions and allocates a block in the backing store the first time a virtual page is read. A subsequent write fault is slightly faster because it requires no further page allocation; it must only mark the page as writable.

9.2 Scalar applications

We made the claim at the beginning of Chapter 9 that the virtualization overhead is often inconsequential. This is not the first time such a claim has been made [75,76]. To further support this claim we ran a number of scalar applications in two scenarios. First, we set up two machines with NFS with the application's executable and data on the server and all execution happening on the client. Although the use of a privileged network file system was a deliberate non-goal for our work, it gave us some basis for comparison of the overhead for distribution of data from file storage to execution domain and back without requiring us to manually copy the data back and forth. Second, we removed the NFS mount and set up a single instance of an MX server on one machine and issued the application from the other machine with an MX client

invocation. Execution occurred on an 8-core 2.67GHz Intel Xeon W3520 running a Linux 3.2 kernel connected to an initiating client system over 1Gb/s Ethernet. In each case, we used a precompiled and unmodified version of the application. Results are shown in Table 9.2.

Our first application is MOCA [77], a 2D full-band Monte Carlo simulator that we ran with its Well-Tempered-MOSFET example. MOCA is a typical scientific application that spends most of its time in uninterrupted execution. It engages in a modest amount of I/O, reading over 9MB of data from six library files and producing 25MB of results in 47 output files.

The second application is the SimpleScalar simulator [78] used by computer architects to model different CPU features. We ran the out-of-order simulator (`sim-outorder`) with the default architectural parameters on the `go.alpha` testcase found in the instructor benchmarks. This application is almost completely computationally-bound. Once the simulator loads its files and establishes a working set, it issues few system calls other than a few writes passed through by the simulated program.

The third application is the SPICE circuit simulator [79]. SPICE does a great deal of floating point computation on sparse matrices. Depending on the complexity of the circuit, SPICE can take a very long time to finish—days are not uncommon. SPICE optimization is an ongoing area of research [80]. We used a modified research version of spice3f5 and ran it with the ram2k benchmark from the ram2k of the CircuitSim90 benchmark suite.

The fourth application we show is BLAST, a Basic Local Alignment Search Tool [81] used to find relationships between nucleotide or protein sequences. We ran BLAST against its extensive benchmark suite [82]. The BLAST benchmark is a complex procedure consisting of a shell script that invokes `make` which repeatedly invokes a Perl wrapper that invokes one of several BLAST executables. BLAST itself is a complicated program that `mmap()`s several database files of 1MB to 218MB in size and scans them in memory rather then doing conventional reads and writes. We

use this as an example of a long-running complex application that requires significant file I/O.

The last application is POV-Ray [83], a popular ray tracing program, to render a short movie. The POV-Ray benchmark consists of a Bourne shell script (using `head`, `grep`, `sed` and `tail` to extract header arguments from a small scene description file) that invokes the renderer to generate 500 frames consisting of PNG files of about 70KB each. It then invokes `ffmpeg` to combine the frames into a single 1MB animation.

Table 9.2
Runtime of MX-virtualized programs compared to native/NFS

Program	Native (seconds)	MX (seconds)	Performance
MOCA	1126.53	1109.87	101.5%
SimpleScalar	501.95	495.72	101.2%
SPICE	1106.12	1115.46	99.9%
BLAST	1390.71	1900.47	73.2%
POV-Ray	113.02	158.87	71.1%

9.2.1 Microbenchmarks

MX is instrumented to count the number of exceptions encountered by an application. By using this facility, we know that this particular run of MOCA experiences 33,833 syscalls and 23,757 page faults. By running microbenchmarks, we also know that MX's syscall time is between 75 and 120 times as slow as a native syscall (about 0.30 seconds additional latency), and its page fault latency is over 60 times slower than native (about 1.42 seconds total additional latency). For MOCA, these additional latencies should manifest themselves as a cumulative 1.7 second longer run time—about 0.1% of the application's total duration. Nevertheless, MOCA in MX

runs slightly faster than native execution. This happens for three reasons. First, the system call latency is often hidden by the transport cost of any data involved. For instance, when the application issues a `read()` system call, the additional latency of the OS virtualization is insignificant compared to the cost of the network operation to fetch the requested data. Second, MX uses prefetching to eagerly pull files to the point of their use. Finally, MX keeps newly created files in the VFS until the application exits. When an application creates and overwrites temporary files, the data for those files are not written out until the virtual session ends. (If the files are deleted, they are not written at all.)

9.2.2 Performance analysis

MX shows nearly native execution performance for the first three applications in Table 9.2 because their system call and page fault load is low. The next two applications involve more pessimistic conditions. For instance, the BLAST benchmark consists primarily of `fork()` and `exec()` calls to invoke new scanners. Each scanner's operations consist of mostly page faults on a memory-mapped file. The POV-Ray run is limited by its many invocations of the `time()` system call. It uses the timing information to show statistical information (even when statistical reports are not requested). Despite the significant overhead of page faults and system calls, these complex applications still run at over 70% the performance of their native equivalent. As we optimize system call and fault handling virtualization these runtimes will improve.

9.3 Interactive programs

Our primary motivation for the development of MX is to enable the use of interactive applications. Two important features are required to make this simple and effective. First, MX has full support for TTYs. This is important for command line editing and also allows us to support general purpose editors and curses-based ap-

plications. Most importantly, it means that interactive signals can be delivered to applications. For instance, if a user is interacting with a shell running in MX, a command launched in that shell can be terminated by pressing `<Ctrl>-C`. This does not terminate the MX client or the session. Instead a SIGINT is delivered to all virtual processes running in that session that share the controlling TTY. Similar behavior happens for `<Ctrl>-\` (send SIGQUIT), `<Ctrl>-Z` (send SIGSUSP), and `<Ctrl>-D` (EOF).

Beyond the traditional Unix shells, the best examples of interactive command-line applications are modern interpreters such as Python [84] or Octave [45]. These applications can be used interactively to develop algorithms, run them, and examine their results. There is no significant difference between execution times of applications running in interpreters that have been invoked natively compared to those invoked through MX. There is, however, often a substantial difference in the initial startup time of the interpreter. Usually, this is only a second or two. Octave is unusual in that it consists of only a 6 kilobyte executable that refers to 58 large shared libraries. The version of Octave we use takes 13 seconds to arrive at the interactive prompt. Once the prompt appears, however, interactivity is indistinguishable from that of a native invocation.

The second important feature we need in order to enable interactive applications in MX is support for graphical user interfaces such as the X11 window system. The MX client can be configured to allow a socket connection to the X display. Internally, MX supports a fully-virtualized TCP and UDP network stack. In the same way that the client can limit access to particular files, it can also limit incoming and outgoing network connections. For instance, the client can designate a particular range of ports to let an application `listen()` for an incoming connection on the native client machine. Unless the client enables external transit of connections, all socket operations are restricted to the isolated virtual network within the MX session.

In addition to running for days or weeks, SPICE is also used interactively. Once a simulation completes, results can be displayed textually or graphically. More inter-

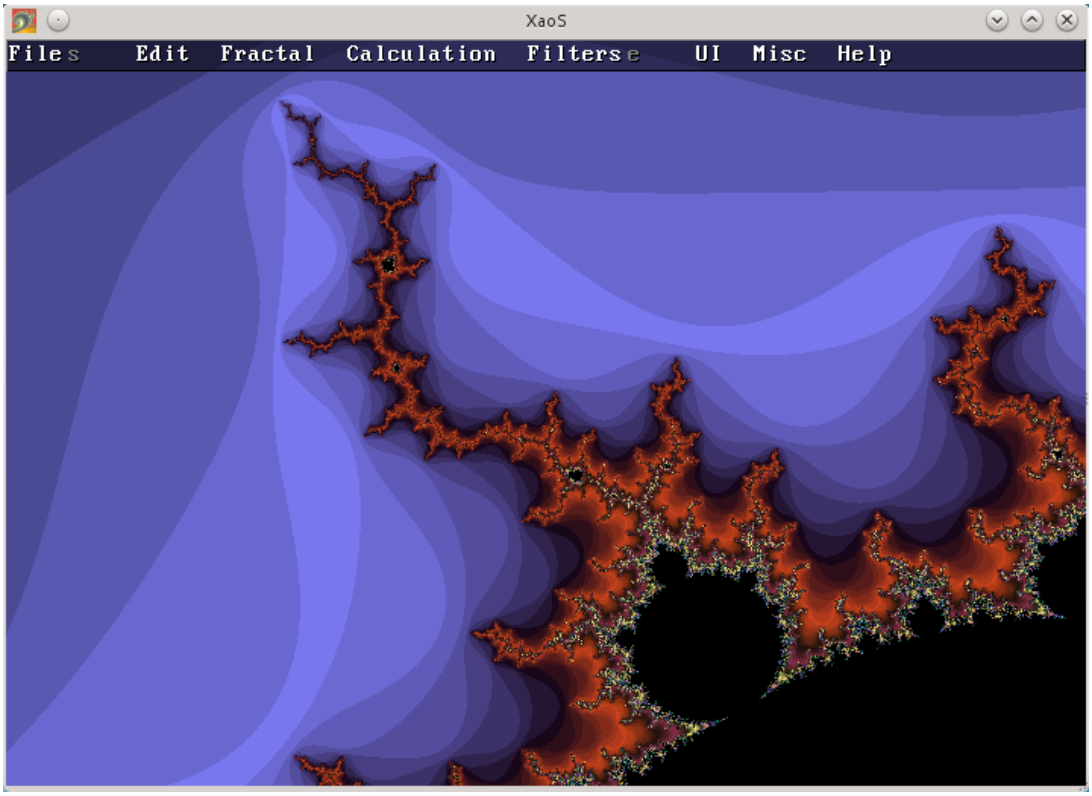

Fig. 9.1. XaoS, a graphical application, running in MX

esting are the applications that allow for direct manipulation of data via a graphical interface. An extreme example is the XaoS fractal viewer [85]. Here, the user can dynamically zoom into a calculated fractal landscape by holding down a mouse button in the desired area. Figure 9.1 shows a screenshot of XaoS running in MX. Speed of interactivity is similar to native invocation.

9.4 Overcommitment

The most direct way to demonstrate a comparison between queue-based scheduling and overcommitment is to create a scenario where applications are invoked at certain times and repeat the scenario for the different situations. Our scenario consists of two full invocations of BLAST benchmark and six short invocations of MOCA (61 seconds

each). To match the conditions of a loaded queue system, we ran the execution host with only one CPU. That way, overcommitment in `mx` would not have an unfair advantage. Results are shown in Figure 9.2.

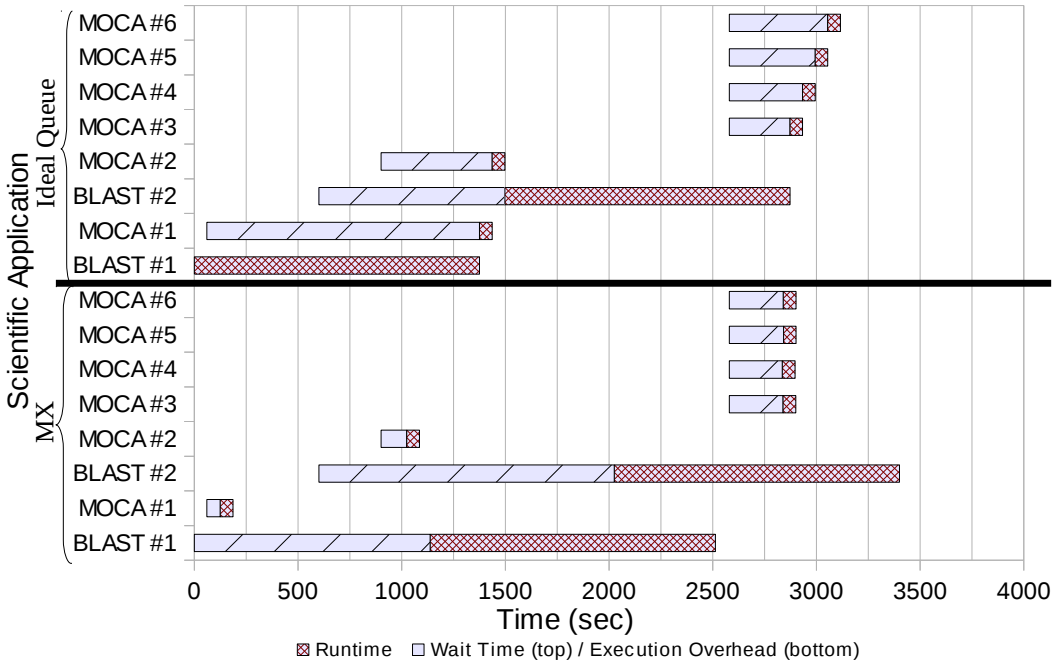

Fig. 9.2. Ideal queue vs. MX

The top graph shows the queue wait and execution times of each application from submission to completion. The bottom graph shows the overcommitment penalty and expected runtime for each application from start to finish. Note that BLAST jobs were delayed due to overcommitment as the short MOCA runs were still moved through in a timely manner. All MOCA runs finished in overcommitment before they would have in a queue. BLAST runs were more significantly delayed. Nevertheless, total completion time for all jobs running overcommitted in `mx` was only 9% worse than completion time for all jobs in an ideal queue. Most of this is simply due to the virtualization overhead penalty currently imposed by `mx` for an application as complex as BLAST.

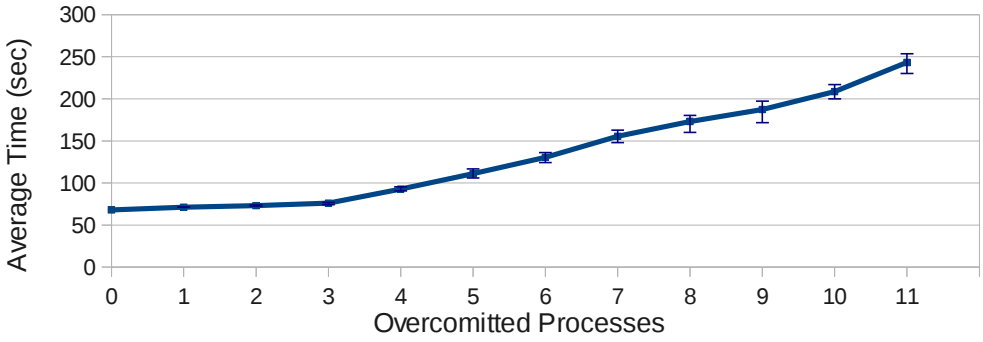

Fig. 9.3. Overcommitment performance degradation

We also ran short MOCA jobs in MX in varying degrees of overcommitment in order to show scalability of overcommitment with `mx`. This time, we used a quad-core machine as a server with multiple clients submitting 61-second MOCA runs through `mx` invocations. Performance degradation occurred in a roughly-linear manner once the four-job machine saturation occurred.

9.5 The impact of migration

To demonstrate the effectiveness of migration for either load balancing or eviction from the execution server, we initiated a 500-second MOCA job from a client to a server through `mx`. While it was running, at random intervals, we forced it to migrate to another slightly slower server machine as if the initial machine was no longer available to serve the load. Once the job finished migration to the secondary server, we immediately migrated it back to the primary machine to avoid skewing the runtime by protracted execution on the slower secondary. Results for zero to ten migrations per run are shown in Figure 9.4. We show here that each incremental migration imposes a linear time penalty on the executing program of about 10 seconds. Some variations in migration penalty occurred due to the number of files that were open by the application at the time as well as the amount of memory it had in its working

set. The user of such an application would likely not have noticed the migration other than a short delay of the output log.

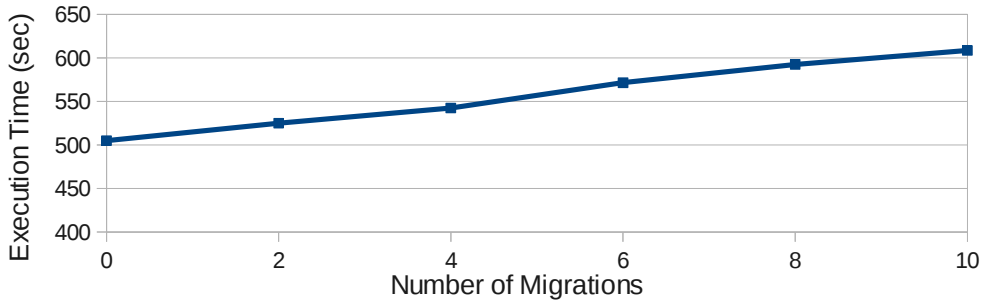

Fig. 9.4. Migration penalty

Process migration is essential for support of interactive applications for two reasons. First, it allows a parallel application to span multiple physical machines by moving virtual processes from one endpoint of an MX session to another. Second, it allows a long-running application to use a short-term resource. For instance, an MX server could be invoked within a traditional queuing system that has a definite time limit for execution. Just before that limit is reached, the MX server can migrate any virtual processes it supervises to another server.

We find that virtualized objects such as tasks, files and vnodes pose negligible overhead in a migration. The time needed to move a task from one system to another is mostly dependent on the quantity of "physical" memory pages referred to by the memory vnode. Table 9.3 shows the time needed to move an application with varying memory size from one machine to another over a 1Gb/sec network. This delay is normally not visible to the end-user since the migrating task is started on the destination before all of its physical pages are moved. Execution lazily faults in the needed pages while migration continues. The table shows that, for smaller memory sizes, each doubling of the size requires double the migration time. For larger sizes, virtualization overhead slows the process of recovery after a migration.

Table 9.3
Migration time for memories of varying sizes

Memory size	Seconds	**Memory size**	Seconds
64MB	0.94	512MB	7.77
128MB	1.83	1GB	19.19
256MB	3.73	2GB	57.10

9.6 Parallel applications

Because MX supports coherent distribution of kernel objects and dynamic migration of tasks, we can use it to distribute parallel applications. Because the use of distributed shared memory over a network would be slow, we would prefer to use a message passing system, such as MPI, to support parallelism. To illustrate this, we used the MPI prime.c example to do an N-way calculation of the highest prime number less than a configurable limit. We compiled this application with the standard MPICH2 libraries and invoked it with an MX client. MX was configured to connect to servers that were started on a cluster of 2.67GHz Intel Core2 Duo systems. Neither the application nor the MPICH2 libraries were installed on these machines. The MX session was configured to do a migrate-on-exec of the MPI multiplexer program. The mpiexec command invokes a multiplexer for each parallel node. MX moves each multiplexer to a different endpoint as it is invoked.

Depending on the calculation limit, the prime application can take many hundreds of seconds. To negotiate the creation of a four-endpoint MX session, invoke the MPI program, and distribute the multiplexers, MX took an additional 6 seconds, regardless of the problem size.

9.7 Simultaneous invocation of applications

We used the same cluster in Section 9.6 to act as a computational farm for running multiple simultaneous instances of our POV-Ray benchmark. An MX server was started on each machine and a client was configured to connect to those servers to run the applications there. On these machines, the run time for a single POV-Ray job is 175.82 seconds, but the MX client uses only 0.4 seconds of CPU time to send and receive data from the run. This makes it possible to invoke multiple MX clients simultaneously without significant impact on the client machine. Table 9.4 (Direct Connection) shows the results of varying numbers of simultaneous runs. In each case, the MX client chose the least loaded servers to use.

Granting a general purpose account on a computation cluster often constitutes more access than an administrator would care to provide. MX servers have no need for a shared filesystem or shared account to run. Furthermore, the student would rather focus on writing and running the application than on setting up the servers. For this reason, MX can be configured to aggregate access to multiple servers through relays. MX has a flexible routing scheme that allows clients, relays, and servers to be interconnected in any topology. To a client, a relay looks just like another server with many more resources. This allows an administrator to simply set up a single relay on the head node to aggregate the servers on an entire cluster. In this way, MX behaves as a Platform-as-a-Service. Table 9.4 (Via Relay) shows the results of routing sessions through a relay.

MX allows the student to easily run an application multiple times with the appearance of local execution while leveraging large distributed computational resources. In less than the time it would have taken to run two POV-Ray jobs locally on the workstation in Section 9.2 (113 seconds each), 32 jobs can be simply distributed via MX to other computers.

Table 9.4
Time to execute simultaneous jobs by direct connection and via a relay

Job Count	Direct Connection (Seconds)	Via Relay (Seconds)
1	175.82	182.79
2	180.57	187.65
4	181.64	189.22
8	185.68	195.38
16	200.81	206.22
32	209.15	238.84

10. CONCLUSION

We have proposed and described a distributed system that is capable of providing transparent client file system projection and simple remote program invocation. This system hides the underlying, complicated mechanics required to distribute execution and as a result has shown to be easy enough for even novice end-users to use.

Through our representative set of programs, we have demonstrated that the system is capable of handling general purpose workloads. In some cases, certain applications actually *benefit* from running inside Metachory.

Our end goal is to have a virtualization system that is generally useful not only to operating system researchers but also to people who want to do practical computation.

10.1 Contributions

The most obvious contribution made by this research is the introduction of an entirely new distributed system. Metachory is a modern system that supports migration for native, unmodified applications. At the time of publication, no other system is capable of supporting execution in this manner. All existing systems require one or more of the following to distribute a captive program:

- Program recompilation or relinking
- Program re-writing to invoke a custom API
- Running a custom kernel, either prepackaged or manually compiled
- Restricting the subset of system calls invoked by the program
- Installation, configuration, and maintenance of a captive OS and associated libraries, binaries, etc

- Extra effort by the end-user to "bootstrap" a program into the execution environment

Metachory has no such requirements.

In implementing this system we have also created a modern distributed coherence mechanism based on the research discussed in [48]. This mechanism is used throughout our kernel and is largely generic.

We have also implemented an advanced form of stack-ripping, called a *context*, that permits opaque data such as pointers to be saved, migrated, and restored through marshalling constructs. Such migration and restoration works even across dissimilar platforms.

These innovations represent the major contributions of this work. There are smaller, novel contributions that arise from our implementation but are small enough to defer discussion to code comments.

This system has great potential not only as a research and teaching tool but also as a practical solution to managing large clusters of machines while simultaneously making them accessible to a much larger userbase.

10.2 Future work

While already composed of over 47,000 lines of code (not including comments and whitespace), our distributed kernel is still very young and remains the subject of intensive development. Much work remains to be done to not only improve its performance but also complete its functionality. In particular, the following items must be completed before the system can practically be deployed in a production environment:

- While a rudimentary checkpoint/restore mechanism exists, it must be cleaned up and generalized to not only be more efficient but also more reliable.
- Additional system calls should be implemented, in turn creating support for more programs.

- A more standardized configuration interface (config file) should be created.

In the long term, more lofty goals include:

- Integrating alternative coherence mechanisms perhaps like in [86]
- Potentially making the server multithreaded.
- Add economic distribution and balancing metrics.

10.3 System availability

Metachory is under active development and will hopefully remain that way for years to come. Source code, documentation, publications, and other errata are available at: `http://metachory.org/`

LIST OF REFERENCES

LIST OF REFERENCES

[1] TOP500 Supercomputer Sites. http://top500.org/, November 2011.

[2] N. Wilkins-Diehr, "Special issue: Science gateways—common community interfaces to grid resources," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 743–749, 2007.

[3] Extreme Science and Engineering Discovery Environment. http://xsede.org/, 2012.

[4] Open Science Grid. http://opensciencegrid.org/, 2012.

[5] M. Litzkow, M. Livny, and M. Mutka, "Condor—a hunter of idle workstations," in *8th International Conference on Distributed Computing Systems*, pp. 104–111, 1988.

[6] Grid Engine Community. http://gridengine.org/, 2012.

[7] S. R. Soltis, G. M. Erickson, K. W. Preslan, M. T. O'Keefe, and T. M. Ruwart, "The global file system: A file system for shared disk storage," *IEEE Transactions on Parallel and Distributed Sytems*, October 1997.

[8] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. Conf. on File and Storage Technologies (FAST '02)*, pp. 231–244, 2002.

[9] Oracle Cluster File System. http://oss.oracle.com/projects/ocfs2/, 2012.

[10] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the Lustre parallel file system," in *Supercomputing*, 2007.

[11] MooseFS network file system. http://www.moosefs.org/, 2011.

[12] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[13] J. H. Howard, "An Overview of the Andrew File System," in *Proc. Winter 1988 USENIX Conference*, pp. 23–26.

[14] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Proc. Summer 1985 USENIX Technical Conf.*

[15] M. Szeredi. http://fuse.sourceforge.net/sshfs.html, 2012.

[16] S. Zhou, "LSF: Load sharing in large-scale heterogeneous distributed systems," in *Wkshp. on Cluster Computing*, 1992.

[17] R. Henderson and D. Tweten, "Portable batch system: External reference specification," tech. rep., NASA Ames Research Center, 1996.

[18] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman, "The Linda alternative to message-passing systems," *Parallel Computing*, vol. 20, no. 4, pp. 633 – 655, 1994.

[19] T. Tannenbaum and M. Litzkow, "The Condor Distributed Processing System," *Dr. Dobb's Journal*, pp. 40–48, February 1995.

[20] F. Tandiary, S. C. Kothari, and A. Dixit, "BATRUN: Utilizing idle workstations for large-scale computing," *IEEE Parallel and Distributed Technology Systems and Applications*, vol. 4, no. 2, pp. 41–49, 1996.

[21] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1994.

[22] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker, "A proposal for a user-level, message passing interface in a distributed memory environment," 1993.

[23] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. Cambridge, MA, USA: MIT Press, 1994.

[24] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, 1990.

[25] D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, (Washington, DC, USA), pp. 4–10, IEEE Computer Society, 2004.

[26] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, and G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM*, vol. 33, pp. 46–63, December 1990.

[27] A. Barak and O. La'adan, "The MOSIX multicomputer operating system for high performance cluster computing," *Journal of Future Gen. Comp. Sys.*, vol. 13, pp. 361–372, Jan 1998.

[28] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch, "The sprite network operating system," *IEEE Computer*, pp. 23–36, February 1988.

[29] F. Douglis, "Transparent process migration in the sprite operating system," Tech. Rep. CSD-90-598, UC Berkeley, September 1990.

[30] D. R. Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, March 1988.

[31] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, no. 2, pp. 95 – 114, 1978.

[32] A. Forin, R. Forin, J. Barrera, M. Young, and R. Rashid, "Design, implementation, and performance evaluation of a distributed shared memory server for mach," in *Proc. 1988 Winter USENIX Conference*.

[33] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser, "Overview of the chorus distributed operating systems," *Computing Sys.*, vol. 1, pp. 39–69, 1991.

[34] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and P. Winterbottom, "The inferno operating system," *Bell Labs Technical Journal*, vol. 2, no. 1, pp. 5–18, 1997.

[35] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proc. 9th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid*, pp. 124–131, 2009.

[36] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Op. Sys. Rev.*, vol. 36, pp. 181–194, Dec. 2002.

[37] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 164–177, 2003.

[38] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "The utility coprocessor: massively parallel computation from the coffee shop," in *Proc. 2010 USENIX Annual Technical Conference*.

[39] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *Proc. 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pp. 339–354, 2008.

[40] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: efficient idle desktop consolidation with partial VM migration," in *Proc. 7th ACM European Conf. on Comp. Sys.*, pp. 211–224, 2012.

[41] D. Williams, H. Jamjoom, and H. Weatherspoon, "The Xen-Blanket: virtualize once, run everywhere," in *Proc. 7th ACM European Conf. on Computer Systems*, pp. 113–126, 2012.

[42] J. Dike, "A user-mode port of the linux kernel," in *Proc. Fourth Annual Linux Showcase and Conference*, October 2000.

[43] H.-J. Höxer, K. Buchacker, and V. Sieh, "Implementing a User Mode Linux with Minimal Changes from Original Kernel," in *Proc. Intl. Linux System Tech. Conf.*, pp. 72–82, 2002.

[44] National Nanotechnology Initiative. http://www.nano.gov/, 2012.

[45] GNU Octave. http://octave.org/, 2012.

[46] *Linux ptrace(2) manual page.*

[47] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: A confined execution environment for internet computations." http://www.cs.ucsb.edu/∼berto/papers/99-usenix-consh.ps, December 1998.

[48] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, pp. 321–359, November 1989.

[49] B. Nitzberg and V. Lo, "Distributed shared memory: a survey of issues and algorithms," *Computer*, vol. 24, pp. 52 –60, aug 1991.

[50] C. Anderson and J.-L. Baer, "Two techniques for improving performance on bus-based multiprocessors," *High-Performance Computer Architecture, International Symposium on*, vol. 0, p. 264, 1995.

[51] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS distributed operating system," *SIGOPS Oper. Syst. Rev.*, vol. 17, pp. 49–70, October 1983.

[52] C. P. Wright and E. Zadok, "Unionfs: Bringing File Systems Together," *Linux Journal*, vol. 2004, pp. 24–29, December 2004.

[53] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management," in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, (Berkeley, CA, USA), pp. 289–302, USENIX Association, 2002.

[54] J. Archibald and J.-L. Baer, "Cache coherence protocols: evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, pp. 273–298, September 1986.

[55] L. Brown and J. Wu, "Dynamic snooping in a fault-tolerant distributed shared memory," in *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 218 –226, June 1994.

[56] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 145–159, May 1985.

[57] S. R. Kleiman and Sun Microsystems, "Vnodes: An architecture for multiple file system types," pp. 238–247, 1986.

[58] *Linux msync(2) manual page.*

[59] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comp. Sys.*, vol. 6, pp. 51–81, February 1988.

[60] IEEE and The Open Group, "The open group base specifications, issue 6, IEEE Std 1003.1," 2004.

[61] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A delegating architecture for secure system call interposition," in *Proc. Network and Distributed Systems Security Symposium*, 2004.

[62] M. B. Jones, "Interposition agents: Transparently interposing user code at the system interface," in *In Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 80–93, 1993.

[63] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *In Proc. Network and Dist. Systems Security Symposium*, pp. 163–176, 2003.

[64] D. A. Wagner, "Janus: an approach for confinement of untrusted applications," Tech. Rep. CSD-99-1056, UC Berkeley, December 1999.

[65] R. N. M. Watson, "Exploiting concurrency vulnerabilities in system call wrappers," in *Proc. 1st USENIX workshop on Offensive Technologies*, pp. 2:1–2:8, 2007.

[66] Subterfugue Project. http://www.subterfugue.org/, 2002.

[67] D. Thain and M. Livny, "Parrot: Transparent user-level middleware for data intensive computing," in *Workshop on Adaptive Grid Middleware*, 2003.

[68] A. R. Butt, C. Gniady, and Y. C. Hu, "The performance impact of kernel prefetching on buffer cache replacement algorithms," in *Proceedings of the 2005 ACM SIGMETRICS international conference on measurement and modeling of computer systems*, SIGMETRICS '05, (New York, NY, USA), pp. 157–168, ACM, 2005.

[69] Y. Artsy and R. Finkel, "Designing a process migration facility: The charlotte experience," *IEEE Computer*, vol. 22, pp. 47–56, 1989.

[70] G. J. Popek and B. J. Walker, "The LOCUS distributed system architecture," 1985.

[71] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable remote execution facilities for the v-system," *SIGOPS Oper. Syst. Rev.*, vol. 19, pp. 2–12, December 1985.

[72] E. R. Zayas, "Attacking the process migration bottleneck," in *In Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 13–24, 1987.

[73] E. R. Zayas, *The use of copy-on-reference in a process migration system.* PhD thesis, Pittsburgh, PA, USA, 1987. AAI8725983.

[74] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Trans. on Software Engineering*, vol. 24, pp. 342–361, May 1998.

[75] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure," *IEEE Computer*, vol. 39, pp. 44–51, 2006.

[76] J. Liedtke, "On μ-kernel construction," in *Symposium on Operating System Principles*, ACM, 1995.

[77] U. Ravaioli, *Monte Carlo Device Simulation: Full Band and Beyond*, p. 197. Kluwer, 1991.

[78] D. Burger and T. M. Austin, "The SimpleScalar tool set, vers. 2.0," *SIGARCH Comp. Arch. News*, vol. 25, pp. 13–25, June 1997.

[79] L. W. Nagel, *SPICE2: a computer program to simulate semiconductor circuits.* PhD thesis, 1975.

[80] N. Kapre and A. DeHon, "Parallelizing sparse Matrix Solve for SPICE circuit simulation using FPGAs," in *Intl. Conf. on Field-Programmable Tech., 2009*, pp. 190 –198, Dec. 2009.

[81] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403 – 410, 1990.

[82] G. Coulouris, "BLAST benchmarks." ftp://ftp.ncbi.nih.gov/blast/demo/benchmark/, 2005.

[83] T. Plachetka, "Pov-ray: Persistence of vision parallel ray tracer," in *Proc. of Spring Conference on Comp. Graphics (SCCG)* (L. Szirmay-Kalos, ed.), pp. 123– 129, 1998.

[84] G. van Rossum, "Python reference manual." http://docs.python.org/reference/, 2012.

[85] GNU XaoS fast interactive fractal zoomer. http://wmi.math.u-szeged.hu/xaos/, 2012.

[86] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in *Proc. of the Winter 1994 USENIX Conference*, pp. 115–131, 1994.

VITA

VITA

Jeffrey Alan Turkstra grew up in Grand Haven, MI. He received the B.S. degree in Computer Engineering from Purdue University in 2004. He received the M.S. and Ph.D. degrees in Electrical and Computer Engineering in 2007 and 2013, respectively, also from Purdue University.

From August 2005 through July 2006, Dr. Turkstra was a Charles C. Chappelle Fellow at Purdue. For the three academic years spanning 2005–2008, he served as an Instructor for the School of Electrical and Computer Engineering. Since June 2009, he has been a full-time Software Engineer with the HUBzero project in the Rosen Center for Advanced Computing at Purdue University.

Dr. Turkstra's research interests are primarily in the areas of operating systems and distributed systems. Previous research activities include a prototype thin client EDA environment based on Sun's GridEngine and Sun Ray Server software as well as a large storage area network (SAN) research project operated by the Engineering Computer Network.

Dr. Turkstra was the recipient of the Graduate Student Award for Outstanding Teaching in 2006 and the Magoon Award for Outstanding Teaching Assistant in 2005.

He also enjoys occasionally dabbling in digital system design and has been working with Microfluidic Innovations, LLC since 2009 on the development of software and hardware control systems for prototype microfluidics chips.

More information is available at Jeff's website, `http://turkeyland.net/`.