Transparent Process Migration in the Sprite Operating System

Frederick Douglis

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

September 1990

Transparent Process Migration in the Sprite Operating System

This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269 and in part by the National Science Foundation under grant ECS-8351961. Additional funding came from the General Electric Corporation and from the California MICRO program.

Transparent Process Migration in the Sprite Operating System

by

Frederick Douglis

Abstract

The Sprite operating system allows executing processes to be moved between hosts at any time. We use this *process migration* mechanism to offload work onto idle machines, and also to *evict* migrated processes when idle workstations are reclaimed by their owners. Sprite's migration mechanism provides a high degree of transparency both for migrated processes and for users. Transparency is ensured by managing shared data structures on a single site and redirecting operations on those structures to the host managing them. Idle machines are identified, and eviction is invoked, automatically by daemon processes. On Sprite it takes up to a few hundred milliseconds on SPARCstation 1 or DECstation 3100 workstations to perform a remote *exec*, while evictions typically occur in a few seconds.

The *pmake* program uses remote invocation to invoke tasks concurrently. Compilations commonly obtain speedup factors in the range of three to six; they are limited primarily by contention for centralized resources such as file servers. CPU-bound tasks such as simulations can make more effective use of idle hosts, obtaining as much as eight-fold speedup over a period of hours.

Process migration has been in regular service for almost two years, used by over 20 day-to-day users of Sprite for nearly all compilations as well as most simulations. Empirical measurements of migration use over periods of time ranging from a month to a year are presented. These measurements include the overall use of migration (31% of all processing in Sprite was performed using migration), the availability of idle hosts (71% of hosts were available for migration during the day, with more hosts available at other times), and the correlation between host idle time and likelihood of eviction (evictions were likely only when hosts that had just become idle were used).

Acknowledgments

I would like to thank several persons for their contributions to this thesis and to my research. My advisor, Professor John Ousterhout, has helped in ways too numerous to mention. His generous support, in terms of time, money, and patience, have helped me to make it through the past many years. His critical attention to my writing has helped to improve my writing skills immensely. He has demonstrated tools for effective management and effective research, and I hope I may carry those tools in the future wherever I go.

The other members of my thesis committee, Professors Arie Segev and Alan Smith, have also provided thoughtful comments that have served to improve both the content and the presentation of my thesis. I appreciate all their efforts, especially their devoting the vast amount of time necessary to read my thesis in the short time they had available.

I would also like to thank those persons who kindly provided many constructive comments on earlier drafts of this thesis: Andrew Cherenson, Thorsten von Eicken, Kinson Ho, Michael Kupfer, and Mendel Rosenblum. They helped me give a much better draft of this thesis to my committee than I would have otherwise.

The past and present members of the Sprite project have helped in many ways. The original members, Andrew Cherenson, Mike Nelson, and Brent Welch, have already left Berkeley. In their time here we went through a lot together, bringing Sprite from an abstract idea to a real-life system we used every day. The later members of the Sprite project, Adam de Boor, Mendel Rosenblum, Mary Baker, John Hartman, Bob Bruce, Ken Shirriff, and Mike Kupfer, made many additional contributions to make Sprite an enjoyable system to use and an enjoyable project to work on. Many of these people were also the original "guinea pigs" who first used process migration on a regular basis. Without their willingness to try new ideas and put up with occasional problems—sometimes interfering substantially for a while with their ability to make progress on their own work—I would never have had the opportunity to make process migration into a viable part of the Sprite system. Adam de Boor deserves special thanks and recognition, since he wrote the original *pmake* program and has always been willing to help me work with it.

The users of process migration in Sprite are too numerous to mention, but Garth Gibson deserves special thanks as the person who has helped push migration beyond compilations and made it usable for other tasks as well. He had to put up with many problems, and I hope the payoff to him, in the form of parallel computation, was worth it.

Finally, I would like to thank my friends and family for keeping me going the past few years. My bridge-playing friends, in particular, have provided vast amounts of enjoyment (and are probably responsible for a collective total of months' worth of delay, alas). My parents have supported me when all else failed. Of course, I cannot begin to thank my wife, Lisa, for all that her companionship and support has meant to me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the past several years, computer use has shifted from relatively slow time-sharing mainframes to higher performance personal workstations, with two results. First, in a collection of personal workstations, many machines are typically idle at any given time. These idle hosts represent a substantial pool of processing power, many times greater than what is available on any user's personal machine in isolation. As this dissertation describes, applications can make use of additional processors by performing tasks in parallel.

Second, users have become accustomed to having a workstation to themselves. If someone else starts using a workstation remotely for a compute-bound or memory-intensive activity, the user sitting in front of the workstation will likely notice degraded interactive response. In the time-sharing machines of the past, large variations in load were inevitable, and users became accustomed to the corresponding variations in interactive response. In contrast, workstation users have come to expect quick, predictable response and are unlikely to tolerate mechanisms that threaten interactive performance.[1] A mechanism to take advantage of idle hosts should therefore also address the preeminence of workstation owners on their own machines.

A wide-ranging set of applications potentially can make use of a facility to execute processes on multiple hosts. Programs with many short, independent tasks (such as compilations) can easily perform work in parallel if multiple processors are available. A user who returns to a workstation that is executing such a small task is unlikely to notice a substantial performance degradation for more than a few seconds, since any "freeloading" processes would complete in a short time. However, many long-running applications could also run efficiently in parallel on separate hosts: for example, simulators are commonly run multiple times with varying parameters, and multiple instances of a simulator could run

[1] On the subject of workstations versus time-sharing systems, the famous one-liner by Jim Morris comes to mind: "The nice thing about an Alto is that it doesn't get faster at night." [Mor]

simultaneously and report their results to a coordinating process. Simulators often take minutes or hours to execute; if a simulator is consuming memory and processor cycles on a host whose owner returns, the owner may be adversely affected for a prolonged period of time.

One possible mechanism for taking advantage of idle hosts is *remote invocation*: programs may be started on other hosts but not moved once they have begun. An example of a remote invocation facility is the Berkeley UNIX *rsh* [Com86] command, which sends commands and their arguments to an intermediary on another host and receives the output of the commands from the intermediary. Though *rsh* is widely available, it is not commonly used to execute commands at the level of small tasks such as individual compilations, because the overhead of invoking *rsh* is high. Each command is executed almost as though the user were establishing a new login session on the remote host. Other commands exist that are similar to *rsh* in functionality but have lower overhead; these facilities include Topaz's "distant processes" [RE87] and SunOS's *rex* [Sun87].

Performance considerations aside, remote invocation does not offer the flexibility or ease of use that is desirable for automatic use of idle hosts in a workstation environment. The greatest problem is workstation ownership: if processes can start on new hosts but not move between hosts once they are started, then a host cannot easily be relinquished once a command has started to execute. Location transparency to the process and to the user are also important. If the environment of the process on a remote host is different from the execution environment on the user's own machine, then the process may behave differently depending on where it executes—or it may have to be specially coded to handle remote execution. For example, *rsh* recreates the user's environment completely from scratch, and *rex* passes environment variables and the current working directory to the remote host but does not provide access to devices on the user's own machine. Neither of these facilities could support remote execution of arbitrary processes. Finally, processes that are created using nontransparent remote execution facilities are isolated from the user: the user's listing of processes will contain entries for a command such as *rsh* but no information about the status of processes on the remote machine. The more remote execution is integrated into the local execution environment, the easier it is for users to take advantage of it.

This dissertation describes an alternate approach to remote invocation, called *process migration*. A process migration facility moves a process's execution site at any time between two machines of the same architecture. Migration allows processes to be offloaded to idle hosts, and it also allows a system to preserve host autonomy by evicting processes from hosts that are no longer idle. Remote invocation is a natural (and commonly used) subset of process migration. To invoke a command remotely, a process migrates to a new host as it replaces its execution image.

For process migration to be useful, it should be *transparent* and *automatic*. Migration needs to be transparent both to the user who runs a remote process and the process itself. With respect to the user, migrated processes should appear as though they were all running on the user's own host: in a listing of processes, rather than seeing several entries for *rsh*, the user might see a number of compilations accumulating processor time. The user could suspend or terminate those processes with no knowledge of the hosts on which they are actually executing. The more important aspect of transparency, however, is its impact on a migrated process. Programs should not need to be coded specially to take migration into account, as long as they are capable of being executed as multiple processes in parallel on a single host. A migrated process should have exactly the same access to system resources as an unmigrated process; if it does not (*i.e.*, if the migration mechanism is non-transparent), then processes may need be restricted from particular operations in order to be able to move between machines. For example, the V System restricts process migration to programs whose output does not depend upon their location [The86], and Remote UNIX migrates only single-process noninteractive jobs that are not permitted to communicate with other processes [Lit87]. Ideally, any process should be able to migrate at any time without affecting its results or the results of other processes.

Process migration should be automatic in two respects. First, load should be spread across idle machines without user intervention. The selection of what processes to migrate and the hosts to which to migrate them can often be performed by the system. For example, a user who types *make* [Fel79] to compile a program need not be aware that compilations are performed on other hosts; the user is only aware that the compilations execute quickly. Second, the system can preserve workstation autonomy by detecting the return of a workstation owner and *evicting* freeloaders (also referred to as *foreign processes*) from the machine, migrating those processes elsewhere. If the processes are evicted quickly and efficiently, the owner may not notice their presence at all.

1.2 Terminology

In order to avoid any potential confusion in terminology, I define several terms that will arise throughout this dissertation. Whenever possible I attempt to remain consistent with terminology used previously in the literature, but there may be some expressions that have no standard definition. Some of these terms have by necessity already been introduced, but they are repeated here for convenience.

process migration: movement of processes from one host to another during execution.

- *source:* host on which process executes at time of migration.
- *target:* host to which process is migrated.

remote execution: execution of a process on a host other than the host used by the person who created the process.

remote invocation: starting a program on a host different from the invoking process.

remote execution facility:	mechanism that supports remote execution via remote invocation and/or process migration.
load sharing:	act of reducing program execution time by distributing processes among multiple hosts rather than executing them on a single host. Load sharing does not imply any particular mechanism, such as migration; it does require support for remote execution.
eviction:	removing a process from a host in order to prevent the process from impacting the host, normally as a result of the host being reclaimed by its owner.
residual dependency:	an on-going need for a process on one host to interact with entities on another host due to remote execution.
remote or *foreign process*:	a process that is executing on a host other than the host on which it would execute in the absence of remote execution.

These terms will be discussed in detail in the next chapter.

1.3 Research Contributions

This thesis describes the design, implementation, and performance of a process migration facility for the Sprite operating system [OCD+88]. The goals of process migration in Sprite are, first, to make the computing power of a collection of workstations available to users as though it were a single, fast time-sharing system; and second, to respect the response-time demands of individual users.

One major contribution of this thesis is the observation that simple methods may be used to migrate processes with high transparency and performance. I examine different methods for transferring processes between hosts and the effects of these methods on transfer time, implementation complexity, and residual dependencies.

A second contribution is an analysis of the interactions between process migration and other components of a distributed system, and methods to manage those interactions. State that is shared by multiple processes on a single host can become distributed across multiple hosts as a result of migration (offsets into open files are an example of this type of state). Also, process migration affects the set of hosts that access files at a given time, which in turn affects the operations needed to ensure consistent file caches. I describe an approach for ensuring consistent access to shared data structures and location-specific resources.

A third contribution is a reevaluation of different techniques for controlling access to idle hosts. The relative advantages and disadvantages of centralized and distributed techniques for finding idle hosts, with respect to simplicity, scalability, performance, and fault tolerance, have been discussed in the literature with no clear-cut resolution (*e.g.*, [BSW89, TL88, The86]). I examine a range of approaches to this problem and conclude that a centralized approach has advantages over distributed approaches in nearly all aspects.

A fourth contribution, and perhaps the most important one, is a demonstration of the effectiveness of process migration in a production environment. I present measurements of the use of migration over a period of up to a year in a community of approximately 20 users. Migration is used automatically for compilations, and some simulators use migration as well. In addition, any command may be executed on an idle host upon the specific request of a user. During the measured period, migration provided approximately an 80% increase in available processing cycles. Evictions have occurred at a rate of approximately 30 per day, demonstrating the desirability of migration above and beyond simpler remote invocation.

Though it has been implemented in several systems, process migration is not yet "taken for granted" by the general public, or by the research community. (It is, however, taken for granted by Sprite users.) This thesis will show that migration has many beneficial effects and can be implemented in a general-purpose operating system without undue complexity. In future years process migration can, and should, become as commonplace as shared network-wide file systems are today.

1.4 Thesis Overview

The next chapter discusses previous work in the area of load sharing. I consider the two primary components of load sharing: the mechanism that supports remote execution and the policy that decides what processes run on what hosts. Most previous work has emphasized one or the other but not both. I present several criteria for the design of load sharing facilities and evaluate existing implementations based on those criteria.

The focus of this thesis is a transparent load sharing facility, using process migration, which I have built in the context of the Sprite operating system. Chapter 3 describes the environment for which this facility was designed: a cluster of personal workstations, many of which are idle at any time, each of which is dedicated primarily to a single user when that user is present. It then describes the overall design of the load sharing facility: the way transparency and autonomy are supported, and the way processes and hosts are selected for remote execution.

Chapter 4 describes Sprite's process migration mechanism in detail. Process migration consists of two interacting phases: process transfer and process execution. The amount of state transferred during the first phase affects both the speed of migration and the speed of process execution during the second phase. Since virtual memory transfer is the aspect of process migration that distinguishes most migration facilities, I compare Sprite's transfer method to those in several other systems with respect to complexity, speed, and reliability. I also describe how Sprite makes processes depend on a single host (called the "home machine") for resources throughout their lifetime, in order to provide a high degree of transparency. On the other hand, a host that evicts processes need not devote resources to the processes after eviction (*i.e.*, processes have "residual dependencies" on their home machine but not on another user's machine).

In Sprite, the greatest complexity in the process migration facility arises from the interaction between migration and the file system. The complexity arises in two ways: the need to support UNIX semantics for shared file streams and deleted files in the face of migration, and the need to keep caches consistent as processes execute on different hosts. Chapter 5 describes how Sprite transfers open files and supports UNIX semantics for files that are shared by processes across a network. It then analyzes the effect of Sprite's cache consistency policy on migration overhead, load sharing performance, and complexity.

The usefulness of load sharing depends not only on the ability to execute processes on remote hosts transparently and efficiently, but the ability to select suitable targets for remote execution as well. In Chapter 6, I discuss possible criteria for determining when a host is available for load sharing, and I compare several techniques for managing the state of hosts in a system and allocating idle hosts to processes. Host selection facilities are distinguished by a number of qualities, including whether the state of each host is stored on a single host or distributed throughout the system and whether decisions about host allocation are made by a single process or by processes on many or all hosts. I conclude that a centralized server process has advantages over other models in the areas of scalability, complexity, and (most importantly) fairness of host allocations.

Chapter 7 reports on the performance of process migration in Sprite. Performance depends on the cost of transferring executing processes and the overhead of supporting transparent remote execution. The overall performance improvement available through load sharing also depends on the extent to which data must be transferred among processes executing on different hosts, as well as any system-wide performance degradation due to increased network and server processor loads. Speedup factors in the range of 3–6 are common for compilations, using from 4–12 hosts. The nonlinear speedup is due to the high utilization of the file server's processor as well as the cost of transferring object files between hosts. However, CPU-bound applications such as simulations do not contend for centralized resources such as a file server or the network, and simulations can therefore obtain much greater performance improvements (8-fold speedups are common).

By observing usage patterns over a period of time, it is possible to validate the design of Sprite's process migration facility and learn ways in which it might be improved. Chapter 8 presents empirical measurements of migration over periods ranging from several weeks to a year. These measurements include the frequency and average cost of remote invocation and eviction; the frequency and success rate of host selection; and the overall processor utilization of the system for both local and remote processes. On the whole, migration has provided about a third of all processing performed in Sprite, with some hosts performing 70–90% of their processing using migration.

Finally, Chapter 9 concludes this dissertation and offers suggestions for future work in the area of process migration.

Chapter 2

Background and Related Work

2.1 Introduction

Previous work in the area of load sharing has typically been divided into two areas, mechanism and policy. The *mechanism* supports remote execution: starting programs on different hosts, managing processes that are running on different hosts, and (possibly) moving processes between hosts. The *policy* determines what processes execute remotely, where they execute, and when they may migrate. Some systems implement primarily a remote execution mechanism but provide little or no support for automatic selection of hosts or processes; for example, the Berkeley UNIX *rsh* command permits a user to execute a specific command on a particular host. Similarly, many studies of policy alternatives have been made without assuming any particular underlying mechanism for providing remote execution. Recently, a small number of systems have combined mechanism and policy into integrated load sharing facilities, and applications have been developed to take advantage of load sharing in these systems.

The dichotomy between policy and mechanism arose from the lack of integration of typical computing environments. Though hosts could communicate via networks, every host had its own disks and its own file system, so the files accessible to a program that was run on one host would not be accessible to it on another. Programs could run on different hosts only if they performed input and output on a restricted set of files (for example, files that are known to be identical on every machine) or they copied all the files they accessed to and from the host on which they executed. The lack of a general remote execution mechanism prevented load sharing policies from being implemented in actual systems, though they were of theoretical interest.

The advent of NFS [SGK+85] and other network file systems dramatically simplified remote execution. Processes on different hosts could now share a similar, if not identical, file system. Programs such as compilers could run equally well on any host that had the same file systems mounted, so a program such as *make* could be modified to execute compilations in parallel on remote hosts. With transparent remote file access, automatic load sharing is much more attainable than with distinct autonomous hosts.

Just because automatic load sharing is attainable, however, does not mean that implementing a system with load sharing would necessarily be worthwhile; that is, would load sharing provide a significant increase in productivity? The potential for improvement arose from the same environmental changes that caused network file systems to become commonplace: the shift from mainframes to personal workstations. Not only does a system with many workstations have much more raw processing capacity than a typical time-sharing computer, but many of the workstations are idle at any given time. Idle hosts are especially common in academic environments:

- Mutka and Livny reported that on average, 70% of workstations in their environment were idle [ML87].
- Theimer and Lantz reported that one-third of the 70 workstations in a particular cluster were completely idle, even during the day [TL88].
- In the Butler system, 50–70 out of 350 workstations were typically in the free pool during the day, with over 100 available machines at night [Nic87].
- My own results, presented in Chapter 8, indicate that 65–70% of hosts in Sprite are idle on average during the day, with up to 80% idle at night and on weekends.

The rest of this chapter is organized as follows. In Section 2.2, I consider some of the trade-offs one might face when designing a remote execution facility, as well as the characteristics that distinguish one design from another. Section 2.3 then describes several existing remote execution and process migration facilities and evaluates them on the criteria presented in Section 2.2. Finally, Section 2.4 describes how load sharing can interact with specific applications.

2.2 Design Considerations

A number of issues arise in the design of a load sharing facility. In many cases, different goals conflict, and one must trade off among several factors. The underlying remote execution mechanism and the higher-level load sharing policy can influence each other considerably. Design considerations include:

- **Transparency.** Remote execution should be location-transparent, so that the physical location of a process is not apparent to either the process itself or the user who invoked it. If execution is location-transparent, then processes should be able to access location-dependent resources, such as the display, time-of-day clock, host name, and so on, without special coding to handle remote execution. Similarly, location-transparent execution permits users to interact with processes uniformly regardless of where the processes execute, so that users need not use special commands to interact with remote processes. For example, a user who performs a remote compile may wish to estimate how much longer the compilation will execute, based on how much processor time it has accumulated. Without a transparent facility, the user might run *ps* on each host, using *rsh*, to list the status of processes on the host. Better yet, with a fully transparent remote execution facility, the user could run *ps* on his own host to determine the same information.

- **Autonomy.** How and when should processes be distributed to other hosts?
  - At one end of the policy spectrum lies the *pool of processors* model. In this model interactive processes and processor-intensive applications are kept distinct. Interactive computation, such as window management, is performed on a different machine for each user. Other computation is performed on a collection of processors shared by all users. On those processors, all scheduling decisions are made automatically by system software. Users submit jobs to the system without any idea of where they will execute, and users do not have priority for particular machines. Amoeba [MvRT+90] and Plan 9 [PPTT90] are examples of this approach.
  - At the other end of the policy spectrum lies *rsh*, which provides no policy support whatsoever. In this model, each host is completely independent. Individual users are responsible for locating idle machines, negotiating with other users over the use of those machines, and deciding which processes to offload. In the normal case, users execute processes only on their own hosts. (Note that *rsh* is an extreme example, which I use to highlight certain issues of transparency and automation; it is not actually an example of "load sharing" in practice.)
  - Process migration is an intermediate point along the policy spectrum. The processing power of the system is not managed as a single common resource, as with the "pool of processors" approach, nor is it managed completely independently by each host, as with *rsh*. With process migration, processes can execute on hosts when they are idle, and migrate elsewhere (or terminate) if a host is reclaimed by its owner.

  Figure 2.1 shows the three approaches just described.

- **Performance.** The performance of a load sharing facility has several components, including the time to select hosts, the time to move an existing process or start a new process remotely, and any performance penalty resulting from executing on a remote host. Performance interacts strongly with many other design considerations, especially transparency and fault tolerance, as described below and in Chapter 4.

- **Residual dependencies.** I have defined a residual dependency as an on-going need for a host to maintain data structures or provide services to a process even after the process migrates away from the host. For example, a host might be required to forward interprocess communication to a process after the process migrates, or the

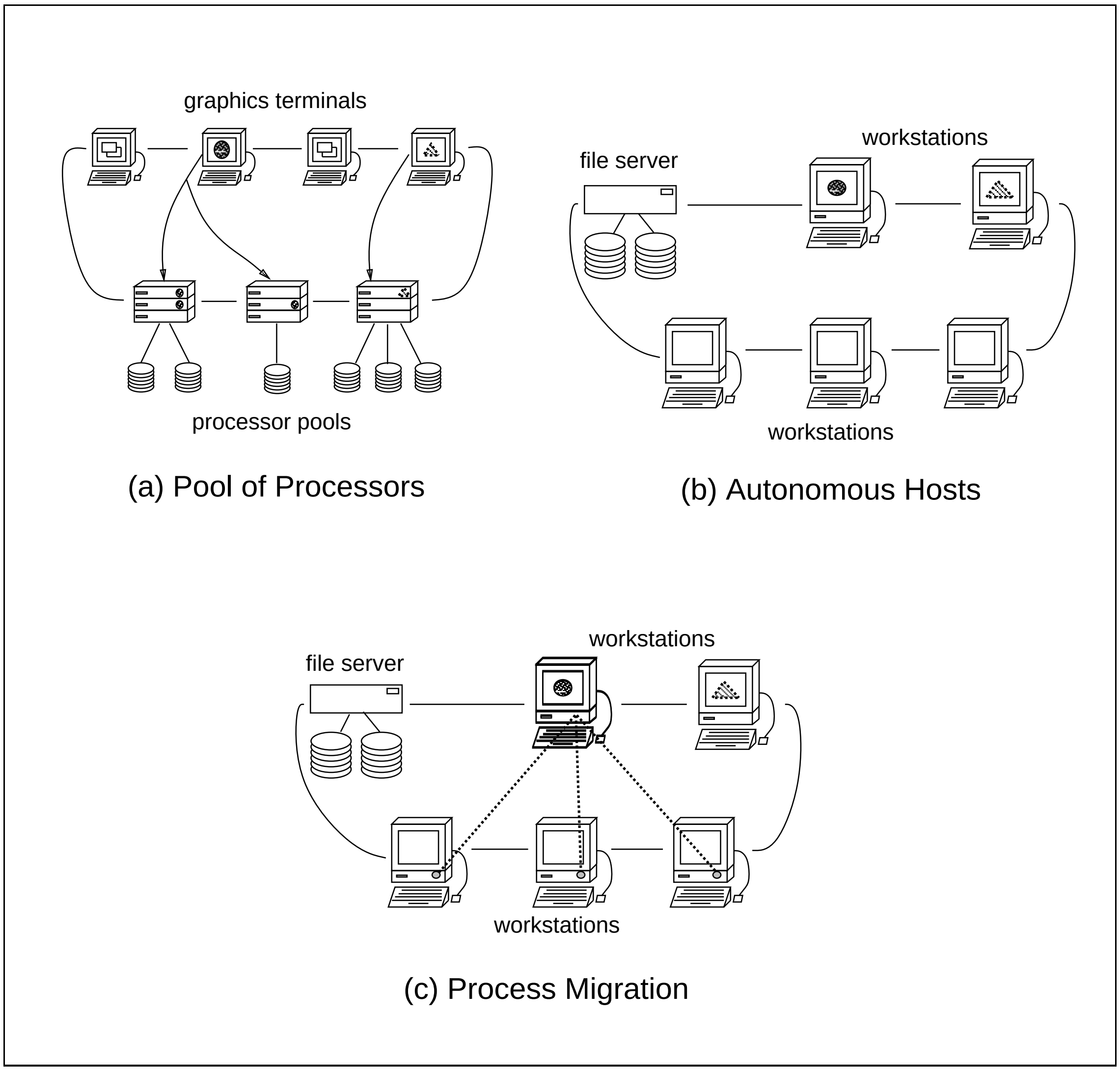

Figure 2.1: *Models for load sharing.* Figure (a) depicts a "pool of processors" approach in which users work on graphics terminals that provide a window system but do no other computation. Shared processor pools are used for most computations. Figure (b) shows the opposite end of the policy spectrum, in which each user runs processes on his or her own workstation, and processes execute completely on a single host unless they are explicitly started on a remote host. Figure (c) is similar to Figure (b), except that processes on one workstation can transparently create and interact with processes on other workstations. With (c), processes can move between workstations in order to preserve workstation autonomy.

process might send requests to the host to access a device. Some residual dependencies are unavoidable: for example, a process that writes its output to the user's display will continue to write to that display (and therefore interact with the user's host) after migration. When residual dependencies may be avoided, one must consider both negative and positive aspects of residual dependencies:

- On the one hand, a process should depend for services only on the host on which it executes. In that case, its performance will not be adversely impacted by communication with other hosts, and the performance of processes on other hosts will not be degraded by requests from the remote process. Additionally, the process will not be vulnerable to the failure of any host except the one on which it executes.
- On the other hand, Zayas has shown that a process with a large address space can migrate away from a host much faster if it leaves its virtual memory pages on the source host and retrieves them later on a demand basis [Zay87a]. In this case, improving performance also increases the migrating process's residual dependencies on the source. If the host with the process's memory image later fails at any time during the process's lifetime, the process might be unable to execute. Increasing transparency also increases residual dependencies, since if a process appears to execute on one host while it physically executes on another, then both hosts must provide support to the migrated process. The failure of either host will affect the functionality of the process.

- **Complexity.** In any system as complex as a load sharing facility, the simpler the system is, the easier it will be to debug and maintain it. In the case of process migration, complexity is a particularly important consideration because process migration tends to affect virtually every major component of an operating system kernel. If the migration mechanism is to be maintainable, it is important to limit its impact as much as possible.

- **Fault tolerance.** A load sharing facility is subject to two types of failures: an inability to select hosts, which would prevent load sharing from occurring, and the failure of a host involved in a particular instance of remote execution. Since a system can function without load sharing, a failure of the scheduling mechanism affects performance but not functionality. The complexity of the facility may be significantly reduced if one attempts to minimize the window of unavailability following a failure, rather than attempting to prevent failures completely. Host failures during remote execution or process migration may be fatal to the processes involved, so one would wish to reduce the number of dependencies that processes have on multiple hosts and/or provide redundancy.

2.3 Implementations of Load Sharing

In this section I describe existing implementations of load sharing facilities. I categorize systems based on the services they provide, their underlying implementations, and their approach toward the factors listed in Section 2.2. Section 2.3.1 describes some facilities that perform load sharing using remote invocation but do not provide process migration. Section 2.3.2 discusses an extension to remote execution, known as *checkpoint/restart*, that permits some types of tasks to be transferred to new hosts. Finally, Section 2.3.3 describes systems that support general process migration.

2.3.1 Remote Invocation

A large number of systems currently provide a form of remote invocation, but few offer support for automatic selection of idle hosts, transparent remote execution, or host autonomy. Maitre d'and Butler provide load sharing facilities using remote invocation, with varying degrees of transparency and autonomy.

Maitre d'

Maitre d' [Ber85] is designed for a collection of time-sharing computers, each with its own file system. Each host typically runs two daemons, one (*maitrd*) that manages exporting tasks during times of high load and another (*garcon*) that accepts remote tasks when the local load is low. Each *garcon* periodically communicates with each *maitrd* in order to keep the *maitrd*s up-to-date about host availability. When a host is loaded, processes on that host use *maitrd* to find a lightly-loaded host (*maitrd* cycles through available hosts in a round-robin fashion). The commands are limited to a pre-determined set of programs, such as compilations and text formatting, that are defined in a system-wide configuration file. To compile Pascal files, Maitre d' runs a special Pascal pre-processor on the remote host that copies files from the local host as they are referenced. Maitre d' uses a different technique to compile C, since much of the work compiling C may be performed using only standard input and standard output. C pre-processing and linking are performed on the local host regardless of load, and compilation and assembly are done remotely. The remote portion of the compilation does not need to access any files but system programs.

Maitre d' is transparent to users to the extent that application programs, such as compilers, can run locally or remotely without user intervention and with the same results. However, users cannot determine the status of their remote programs, or even learn where those programs are being executed, and only a small set of programs are suitable for remote execution using Maitre d'. Maitre d' grants local processes priority over foreign processes by reducing the execution priority of foreign processes when the machine's load passes a threshold. This policy has the effect of silently suspending the process until the load is reduced, since the process receives little or no share of the processor. Since host selection is distributed, the failure of one host, or one daemon, does not affect the ability of other hosts to select hosts or run remote commands. Bershad found that over a two-month interval, Maitre d' provided more predictable response times and somewhat shorter execution times compared to a system with no load sharing. For example, the time to start up an editor had a variance of .362 seconds when Maitre d' was used for compilations, compared to a variance of 1.87 seconds without Maitre d', and the mean compilation time dropped from 11.9 seconds to 7.04 seconds (with a drop in its variance from 94.6 seconds to 20.42 seconds).

Butler

Butler serves as an agent to manage resources in a collection of workstations. The Butler system has had two incarnations, one as a research prototype and one that has received considerable use. Dannenberg's prototype [Dan82] was built on the Accent system and used process migration to reclaim hosts; process migration in Accent is described below in Section 2.3.3. Nichols [Nic87] reimplemented Butler for the Andrew system, making no kernel modifications and requiring no changes to application programs.[1] The decision to avoid kernel modifications limits Butler's functionality in exchange for simplicity and portability. Process migration is not provided, so when hosts are reclaimed, any foreign process on it is terminated (after notifying the process and the user running it).

Butler is used upon the explicit request of a user: the user specifies a command to be executed, and Butler locates an idle host and executes the command on that host. Since the remote program can be a command interpreter, it is common for users to execute many commands remotely in a single interactive session. Nichols noted that the overhead of starting a program remotely is substantially greater than invoking a program from a remote shell, so using a long-running remote shell amortizes the cost of remote execution. In addition, file caching effects make repeated use of the same machine more desirable than using a random sequence of machines over time.

Butler is reasonably transparent and can support most programs without modifications. It is not completely transparent, however, since the execution environment varies slightly from host to host: for example, each host has its own directory for temporary files. Since Butler respects workstation autonomy by terminating foreign processes, it is not as convenient to use as a system that migrates processes instead. (Nichols noted that the ability to migrate existing processes would make Butler "much more pleasant to use.")

Host selection is performed in Butler using a centralized machine registry (*mreg*) process, which receives periodic information from a daemon on each idle host. If the *mreg* process terminates, one of the daemons reinstantiates it on an idle host. Including the overhead to select a host and start a process on it, remote invocation using Butler takes several seconds, indicating that Butler is not suitable for starting individual processes unless they are expected to execute for a long period of time.

[1]Unlike the Accent prototype Butler, the Andrew Butler has been used extensively; unless specified otherwise, the term *Butler* refers to Nichols's version for the remainder of this thesis.

2.3.2 "Checkpoint/Restart"

If the system can checkpoint the state of a process and create a new process with that state, it may provide a restricted form of process migration. Migration would be "restricted" because the new process would not have the same process identifier or parent process, and it might not have the same access to network connections or other open files.

Smith/Ioannidis

Smith and Ioannidis [SI89] implemented a variant of the *fork*() system call that would create an executable image of a process on a new machine using checkpoint/restart. The executable image would be copied to another host using a standard file-copying program such as *rcp*. To "migrate" itself, the parent could *fork* a child process onto a new host and then exit, leaving the child running remotely.

In this system, the checkpoint contained no information about such state as open files, current working directory, or parent/child relationships. In addition, the system was implemented on standard UNIX using NFS, so execution was not location-transparent—each host had its own file system name space and local devices, for example. Thus, this form of migration would be suitable for compute-bound operations but not for processes that use files, unless the processes were prepared to reopen the files subsequent to each migration. This model of checkpoint/restart does have the advantage that a process depends only on its current host for resources, so processes can move from a host that is about to be shut down. It is also relatively simple to implement.

Alonso/Kyrimis

Alonso and Kyrimis [AK88] also implemented a checkpoint/restart facility on top of UNIX, with changes to the kernel to support fast dumping and restoring of kernel data structures and to maintain additional information about names of open files. Unlike Smith and Ioannidis's implementation, the state of most open files is restored when a process is restarted. (Pipes and sockets cannot be reopened.) As with the other implementation, processes cannot depend on their execution environment, such as their process identifiers or parent/child relationships.

Remote Unix and Condor

Remote UNIX is a remote execution facility for UNIX, suitable for background CPU-intensive applications [Lit87]. It requires no modifications to the UNIX kernel, instead using a special run-time library to forward nearly all system calls to a surrogate process running on the host that initiated the remote process. As in Smith and Ioannidis's system, processes can be checkpointed and resumed elsewhere. The checkpointed state includes virtual memory, registers, and open files, but not other process-specific state such as network connections or parent-child relationships. The initial implementation of Remote UNIX also did not support several system calls, including *fork*(), *exec*(), and network-related calls, though those were to be added.

Condor [LLM88] uses Remote UNIX to make use of idle workstations. Tasks are submitted to Condor using a "batch" mode; *i.e.*, no interaction with the user is permitted, and the tasks are queued for execution if no host is immediately available. Litzkow, Livny, and Mutka measured the use of Condor over a one-month period to determine usage patterns and execution overhead. They found that nearly all tasks using Condor executed for more than an hour, using almost 200 machine-days of processing time on 23 workstations. Condor uses the Remote UNIX checkpoint/restart mechanism to move these long-running processes off of hosts that are reclaimed by their owners. If no host is available, processes are suspended and queued for later execution.

The combination of Remote UNIX and Condor is a powerful and simple tool but has a limited domain (long-running, compute-bound batch jobs). Remote execution is transparent to processes using the facility, and because all processes are run in batch mode, users are not expected to interact with the processes. (Presumably Condor provides a mechanism for users to determine the execution status of background jobs, and to terminate them.) On the other hand, only programs that have been specially linked with the Remote UNIX run-time library can run remotely.

Condor meets the criteria described above in Section 2.2 in many respects. It provides autonomy by checkpointing remote processes, permitting processes to be terminated or moved elsewhere when hosts are reclaimed. It is reasonably fault-tolerant: it uses a centralized scheduling facility that obtains information from daemons on each host, and if the centralized facility fails it can be reinstantiated quickly on another host. However, remote processes have residual dependencies on the host running their surrogate process, and the failure of that host will terminate the remote process. The performance of Condor depends on the processes that use it, since the need for a large number of forwarded system calls makes Remote UNIX unsuitable for I/O-intensive processes. For compute-bound jobs, Litzkow, *et al.*, found that the overhead of remote execution was extremely low: only one minute of local processing was needed to support nearly a day of remote processing time.

2.3.3 General Process Migration

In addition to the aforementioned systems that provide processes with the ability to start processes on different hosts, a number of systems permit processes to move during execution. Process migration is a generalization of remote invocation, since any system that performs migration provides remote invocation (a process forks a child, which migrates and *exec*s a new execution image). Compared to remote invocation, process migration provides the additional flexibility of permitting hosts to be reclaimed without the need to terminate foreign processes. Also, systems that support process migration tend to provide a higher degree of location transparency to remote processes than do systems that support only remote invocation.

Systems that support process migration can be categorized in two ways: message-passing systems or kernel-call-based systems. Most existing implementations of process migration are in message-passing systems (such as V, Charlotte, or Accent), where all communication between a process and the rest of the world occurs through message channels. In these systems, many of the transparency aspects of migration can be handled simply by redirecting message communication to follow processes as they migrate. In UNIX-like systems, such as LOCUS, MOSIX, or Sprite, system calls and other forms of interprocess communication (IPC) are invoked by making protected procedure calls into the kernel. In such a system the solution to the transparency problem is not as obvious; in the worst case, every kernel call might have to be specially coded to handle remote processes differently than local ones. In the following sections I describe how existing systems provide transparency, and what special measures they take (if any) to improve the performance or general usability of migration.

V System

Theimer implemented process migration in V [The86, TLC85], a message-passing system, with an emphasis on reducing the time during which a process is moving between hosts and unable to execute. This "freeze time" was important for two reasons that were particular to V: first, it did not support virtual memory, so a process's entire address space had to be copied to the memory of the target host during migration; and second, messages to suspended processes could time out and result in errors. Since suspending a process for the entire time needed to copy its address space would cause delays sufficient to generate time-outs on messages to the process, Theimer implemented *pre-copying*: a process is allowed to execute while its address space is transferred. If pages are modified after being copied to the target, they must be sent again. The procedure iterates until a small number of pages are dirty, at which point the process is frozen and the rest of its state is transferred (including the remaining dirty pages). Thus, the time during which a process is frozen is minimal, but by comparison to a system that transfers memory monolithically, the total time taken to migrate a process is increased because of the pages that must be transferred twice. The complexity of the system is increased as well.

The V System remote execution facility is nearly completely transparent. Processes communicate via a network-transparent interprocess communication facility, so the physical location of a process does not affect the ability of other processes to interact with it. Processes themselves execute equivalently when running remotely as locally, with a small number of exceptions; for example, if a remote process explicitly requests information about the host on which it is executing, it is given information about its current host rather than the host from which it was invoked.

V supports *multicast*, which provides an inexpensive means of communicating with several machines simultaneously. Theimer used multicast to simplify the task of locating processes after migration. Kernels cache the most recent known location of a process, but if they find that a process is no longer at the same location, they multicast to find the process's new site.

Charlotte

In Charlotte [AF89], another message-passing system, Artsy and Finkel designed migration to be as fault-tolerant as possible. Migration can be aborted any time before migration reaches a commit point (the time at which the process's address space and message channels are transferred). After migration, processes leave no residual dependencies, such as forwarding pointers, on the source; therefore, the failure of a machine on which a process had once executed does not affect the process unless it is communicating with other processes on that host. By eliminating residual dependencies, Charlotte permits migration to be used not only for host autonomy but also for fault tolerance, since processes can migrate away from a host that is about to fail.

Charlotte message channels are bidirectional, and kernels know the location of the process at the other end of any channel. When a process migrates, the kernels that have message channels to it are notified of its new location. The requirement of immediate notification simplifies the implementation but degrades migration performance. As in V, processes are accessed via communication links that are independent of location, so the act of migration is transparent.

Accent

Dannenberg's Butler was a prototype remote execution facility that used process migration to transfer processes between hosts [Dan82]. In this version, virtual memory was transferred monolithically. Zayas modified Accent to migrate processes using copy-on-reference virtual memory [Zay87a, Zay87b]. When a process migrates, its memory image is initially left on the source machine; only the process's page tables, registers, and message channels need be transferred immediately. As the process executes, it demand-pages its memory from the source. Zayas found that over one set of benchmarks, migration using copy-on-reference virtual memory eliminated 21% to 96% of data transfers, compared to copying memory directly.

Copy-on-reference memory trades migration time for increased complexity and residual dependencies. Implementing copy-on-reference network-wide memory was complicated by the need to add a new class of virtual memory object and the need to redesign and reimplement the Accent network-wide message server. Accent's process migration facility does not fully address the issue of host autonomy, since a user's workstation would continue to service requests after a process was evicted. It also makes a process vulnerable to the failure of all the hosts on which it has executed, since a process that migrates several times could potentially reference memory on each host.

Accent provides name and location transparency to processes by means of message channels. It does not provide automatic load sharing, though Butler could easily be reimplemented on top of the current Accent process migration facility.

LOCUS

LOCUS [PW85] is a UNIX-like system that provides two forms of migration, one for transferring a process at an arbitrary time and one for performing an *exec* on another host. By treating *exec*-time migration specially, LOCUS provides fast migration in the common case when a process's address space does not need to be transferred. When the virtual memory of a process must be transferred, the entire data segment is copied from the source to the target; the code segment of a migrating process normally does not need to be transferred, since a copy may already exist on the target and otherwise can be demand-paged from the host that stores the code.

In LOCUS, process identifiers include an identifier for the host on which a process is created, known as the *origin site*. That host is guaranteed to know the current location of the process. The origin site serves as a rendezvous point when other kernels need to manipulate the process. For example, to send a signal to a process, the kernel delivers the signal locally if the process is on the same host as the kernel, and it obtains the process's current execution site from the origin site if not. In addition, the origin site manages information regarding process termination. If the origin site fails, another host serves as a surrogate origin site until the origin site becomes available again.

In UNIX-derived systems, processes inherit file descriptors from their parents and share a single access position into those shared files. If process migration causes a file access position to be shared between processes on different machines, then input or output by a process on one machine must be reflected in the access position seen by processes on other machines. LOCUS lets the sharing machines take turns managing the access position. In order to perform I/O on a file with a shared access position, a machine must acquire the "access position token" for the file. While a machine has the access position token it caches the access position and no other machine may access the file. The token rotates among machines as needed to give each machine access to the file in turn. This approach is similar to the approach LOCUS uses for managing a shared file, where clients take turns caching the file and pass read and write tokens around to ensure cache consistency.

LOCUS remote execution is partially transparent, since process identifiers and file names are globally unique, but processes execute in the environment of the machine on which they reside. For example, they appear in the listing of active processes on that machine, and they will obtain host information for that machine rather than their origin site.

The AIX Transparent Computing Facility [WM89], which derived from LOCUS, supplies some programs to help users select machines for remote execution. It allows users to specify that programs should be run on lightly-loaded hosts, or for daemons to migrate processes for the purposes of smoothing the load across multiple hosts.

MOSIX

MOSIX [BS85, BBNG$^+$89, BSW89], also a UNIX derivative, uses a single paradigm to move processes both between hosts and between processors within a multiprocessor. Intrahost migration is performed because processors are not treated symmetrically: there is a single master processor and a variable number of slave processors, each running a distinct kernel. Any reference by a process to a kernel resource includes information about the kernel that manages the resource; when necessary, system calls are forwarded from the processor on which a process executes to the kernel controlling the resource.

MOSIX implements the "pool of processors" approach to load sharing, and process migration is performed for the purpose of load balancing rather than host autonomy. Processors in MOSIX share load information by trading load information with other processors chosen at random. Load information is cached in a fixed-length vector, so each processor has information about only a few other processors at any time. Each time new load information is obtained, the kernel decides whether to migrate a process elsewhere to maintain a balanced load. The MOSIX load balancing policy is fairly complex. For example, kernels adjust the load they report in order to account for relative processor speeds and anticipated fluctuations in future loads. This helps to avoid migrating processes too often, and it is necessary to cope with the sudden creation of many new processes on a processor.

2.4 Applications

With the recent wide-spread availability of multiprocessors and load sharing facilities, several multi-process applications have developed. The most common application to make use of load sharing in existing systems is a program to perform compilations in parallel. In a typical UNIX system, parallel compilations are characterized by the execution of many independent, short-running processes. As the availability of hosts changes over time, the number of executing processes can change as well. Other parallel applications typically involve long-running processes that cooperate to perform a single task. Some parallel applications that are well suited for execution on a multiprocessor are also suited for a distributed system with transparent remote execution. Their ability to execute over a slower network depends primarily on their requirements for interprocess communication. The next sections consider existing applications that could take advantage of load sharing.

2.4.1 Parallel compilation

Make [Fel79] is a UNIX program that allows users to specify commands to bring files up to date and dependencies between files. It is commonly used to recompile programs when some of their source files have changed, though it may be used for other purposes as well. For compilations, each command typically takes several seconds to execute, though one invocation of *make* might execute hundreds of commands. The standard *make* program that is distributed with Berkeley and AT&T UNIX performs operations sequentially.

Several people have modified or rewritten *make* to execute independent commands in parallel, on multiprocessors and/or loosely-coupled hosts in a distributed system. Typically, parallel versions of *make* will execute as many commands simultaneously as there are processors available. If *make* checks periodically on the availability of processors, it can increase or decrease its parallelism as the need arises.

The performance improvements from parallel *make* that have been reported vary considerably. As reported by Fleckenstein and Hemmendinger, Encore has obtained linear speedup using four processors in a shared-memory multiprocessor, with a speedup of 11 using 16 processors. Fleckenstein and Hemmendinger themselves implemented a parallel *make* using Linda and NFS, obtaining a speedup of two to three using four hosts, with minimal improvement beyond four hosts for their benchmark [FH89]. Baalbergen showed a speedup of 3.5 to compile the same file 4 times in parallel on separate Amoeba hosts [Baa86]. Roberts and Ellis have obtained speedups ranging from 6 to 12 using 15 machines, with the limiting factor being disk traffic to and from the controlling machine [RE87].

2.4.2 Distributed Applications

Barak, *et al.*, converted several single-machine programs to take advantage of MOSIX's dynamic load balancing by executing in parallel on a multiprocessor. They implemented distributed versions of the traveling-salesman problem, several simulations, and image processing tasks. The traveling-salesman problem obtained a speedup that was nearly linear in the number of processors, while other applications showed less speedup due to communications overhead [BBNG+89]. (It is important to note, however, that these speedups were obtained on shared-memory multiprocessors, and that comparable measurements on a loosely-coupled distributed system might be affected more strongly by communication costs.)

2.5 Summary

By distributing processes across multiple hosts, a system can substantially reduce the time needed to execute a variety of programs. However, load sharing facilities need to address several concerns beyond performance. My primary goals for process migration in Sprite have been ease of use and preserving host autonomy. Most existing systems restrict the class of processes that can execute on remote hosts, or do not provide the same environment to remote processes as to local ones. Users may have to interact with remote processes using special commands or interfaces. Only a few systems can move running processes in order to reclaim hosts. In contrast, I wish to make a collection of workstations as easy and efficient to use as a single time-sharing system with many processors, while still guaranteeing individual users the processing power of one workstation. The next chapter describes how this is done in the Sprite operating system.

Chapter 3

Load Sharing in Sprite

3.1 Introduction

As the previous chapter shows, there are many different ways to implement and use a load sharing facility. For example, the goals of a load sharing facility for time-sharing computers will likely be different from the goals of one that runs on personal workstations. The underlying operating system may differ radically from one facility to another, affecting the feasibility and complexity of transparent remote execution. A facility may be used for only long-running noninteractive processes, such as Condor is [LLM88], or it may be intended for a mixture of interactive and noninteractive processes with varying needs for resources. The particular environment influences the design of the facility, within the spectrum of alternatives described in Chapter 2.

Section 3.2 describes the environment in which the research reported in this dissertation has been conducted. It discusses the Sprite operating system and its implications for load sharing. Section 3.3 presents the design of a transparent load sharing facility for Sprite using process migration. Section 3.4 summarizes load sharing in Sprite.

3.2 Sprite

Sprite is an operating system for a collection of personal workstations and file servers on a local-area network [OCD+88]. Sprite's kernel-call interface is much like that of 4.3 BSD UNIX [Com86], but Sprite's implementation is a new one that provides a high degree of network integration. Each host runs a distinct copy of the Sprite kernel, but the kernels work closely together using a remote-procedure-call (RPC) mechanism [Wel86] similar to that described by Birrell and Nelson [BN84].

All the hosts on the network share a common high-performance file system [Nel88, Wel90]. Sprite permits the data of a file to be cached in the memory of one or more machines, with file servers responsible for guaranteeing "consistent access" to the cached data. A file server keeps track of which hosts have a file open for reading and writing. If a file is open on more than one host and at least one of them is writing it, then caching is disabled: all hosts must forward their read and write requests for that file to the server so they can be serialized.

Because servers need to maintain state about all open files in order to ensure consistent caches, process migration and the file system interact strongly. When a process changes hosts, the servers that control access to its open files must be notified about its new location. The change of host may cause a cached file to become uncachable, or an uncached file to become cached. Chapter 5 discusses the interaction between the file system and process migration in detail.

Though the file system is physically distributed, it is logically centralized because all hosts share a single name space. The centralized file system is used for virtual memory paging and all interprocess communication (IPC). As I discuss in the next chapter, paging via the file system simplifies migration because the functionality to demand-page a process over the network already exists. Sprite's interprocess communication paradigm also simplifies migration considerably. Processes communicate using a special type of file known as a *pseudo-device* [WO88]. The migration of a process is transparent to the processes with which it communicates, because only the operating system stores the location of the processes that use the pseudo-device. Even communication using Internet (IP/UDP/TCP) protocols over sockets is passed via a pseudo-device to and from a server process [Che87], so Internet socket IPC does not pose any particular problem for migration.

3.2.1 Considerations for Migration Design

Several other aspects of the Sprite environment were particularly important in the design of Sprite's process migration facility:

Idle hosts are plentiful. Since our environment consists primarily of personal machines, it seemed likely to us that many machines would be idle at any given time. This belief was supported by the studies reported in Section 2.1; later, my own measurements (given in Chapter 8) showed 65–80% of all workstations are idle on average, even during weekdays. The availability of many idle machines suggests that simple algorithms can be used for selecting where to migrate: there is no need to make complex choices among partially-loaded machines.

Users "own" their workstations. Prior to designing the Sprite process migration facility, I surveyed our potential user community about their reaction to sharing processor cycles, both in UNIX and eventually in Sprite. Though people favored the opportunity to improve performance, the bias toward "ownership" of workstations was clear: they were unwilling to permit other users' processes on their machines while they were active, even if the processes would run at low priority or for only a short time, because the foreign processes would compete with their processes for resources such as physical memory. This suggested that a machine should only be used as a target for migration if it is known to be idle, and that foreign processes should be evicted if the user returns before they finish.

Most programs are short-lived. In a UNIX environment, most of the programs that users invoke interactively execute for short periods of time, while a small number of processes execute for long periods. For example, Zhou traced a VAX-11/780 4.3 BSD UNIX system and found that the mean process execution time was 1.5 seconds with a standard deviation of 19.1 seconds [Zho87]. We presumed that the same pattern would hold in Sprite. Under this assumption, migrating active processes will provide significant performance improvements only if the overhead of migration is extremely low (a few hundred milliseconds at most) or migration is limited to processes that are known to be long-running.

Sprite uses kernel calls. As noted in Section 2.3.3, most other implementations of process migration are in message-passing systems, in which transparency is provided primarily by redirecting communication. Sprite processes perform protected procedure calls into the kernel on the host on which they execute. To provide location transparency, the kernels must cooperate by explicitly forwarding calls between hosts, rather than by simply redirecting message channels.

Sprite already provides network support. I was able to capitalize on existing mechanisms in Sprite to simplify the implementation of process migration. For example, Sprite already provided remote access to files and devices, and it has a single network-wide space of process identifiers. These features and others made it much easier to provide transparency in the migration mechanism. In addition, process migration uses the same kernel-to-kernel remote procedure call facility that is used for the network file system and many other purposes. On SPARCstation 1 workstations (roughly 10 MIPS) running on a 10 megabits/second Ethernet, the minimum round-trip latency of a remote procedure call is about 1.6 milliseconds and the throughput is 480-660 Kbytes/second. Much of the efficiency of the Sprite migration mechanism can be attributed to the efficiency of the underlying RPC mechanism.

3.3 Load Sharing Design

Two conflicting goals motivated the design of Sprite's load sharing facility. On the one hand, we wanted transparency: Sprite should present the illusion of a single fast time-sharing system, rather than a distributed system with many independent hosts. A user performing a large parallel compilation on one workstation should not be aware that processes are being executed on many machines at once, only that many processes are executing. On the other hand, we needed autonomy: we were unwilling to give all users equal access to all resources the way a time-sharing system would. The need for users to have "eminent domain" over their workstations was compelling. We wished to use workstations for load sharing only when they were idle, and to have a way to reclaim workstations automatically when they were no longer idle. Furthermore, when a workstation is reclaimed by its owner, it should not have to dedicate any resources to processes that had been running on it; *i.e.*, after eviction, a migrated process should have no residual dependencies on the host that evicted it.

Section 3.3.1 describes our approach to transparent process migration, which we viewed as a way to address both of these goals. Section 3.3.2 presents an overview of Sprite's load sharing policy. Both of these subjects are discussed in detail in subsequent chapters.

3.3.1 Transparent Process Migration

Sprite presents the illusion of a time-sharing system by making a process appear to execute in a single location throughout its lifetime. That location is referred to as the *home machine* of the process; it is the machine where the process would have executed if there had been no migration at all. A *remote process* (one that has been migrated to a machine other than its home) has exactly the same access to virtual memory, files, devices, and nearly all other system resources that it would have if it were executing on its home machine. Furthermore, the process appears to users as if it were still executing on its home machine: its process identifier does not change, it appears in process listings on the home machine, and it may be stopped, restarted, and killed just like local processes. In the next chapter, Section 4.3 describes in detail how transparent remote execution is supported in the kernel, and Section 4.4 describes how Sprite eliminates residual dependencies when processes are evicted.

3.3.2 Policy

In Sprite, kernels implement the process migration mechanism while user-level applications implement the load sharing policy. User-level support takes four forms: idle-host selection, eviction, the *pmake* program, and a *mig* shell command.

Selecting Idle Hosts. Each Sprite machine runs a background process called the *load-average daemon*, which monitors the usage of that machine. When the workstation appears to be idle, the load-average daemon notifies the rest of the system that the machine is ready to accept migrated processes. Processes that wish to use idle hosts obtain the hosts via a library call and return the hosts to the pool of idle hosts upon completion. This "check-in/check-out" procedure avoids instabilities that might result from multiple processes simultaneously migrating to the same host. Alternative solutions are discussed in Chapter 6.

Eviction. The load-average daemons detect when a user returns. On the first keystroke or mouse-motion invoked by the user, the load-average daemon will check for foreign processes and evict them.[1] Processes return to their home machine, and can be remigrated from there if another host is available.

Pmake. The most common use of process migration is by the *pmake* program, which is similar to the parallel-compilation programs described in Section 2.4.1. *Pmake* uses process migration to invoke as many commands in parallel as there are processors available, either locally or on idle hosts. It also handles evictions by remigrating evicted processes or temporarily suspending them until a new host is available.

Mig. The *mig* program will select an idle machine using the mechanism described above and use process migration to execute a specified command on that machine. *Mig* may also be used to migrate an existing process.

3.4 Summary

The design of Sprite's load sharing facility was influenced by several factors: our workstation environment, typical applications that use Sprite, and Sprite's underlying implementation. The workstation environment implied the need for autonomy as well as the opportunity for a simple approach to host selection, due to the wide availability of idle hosts. Sprite's network-transparent kernel permitted me to implement a simple process migration mechanism with full transparency.

Transparency in Sprite is defined with respect to the home machine of a process. A process appears to execute on its home machine throughout its lifetime, and its behavior is unchanged in the presence of migration. It has residual dependencies on its home machine if it executes elsewhere, since the home machine is responsible for knowing the process's location, but after eviction a process has no residual dependencies on the host from which it is evicted.

Load sharing in Sprite is used primarily by *pmake*, and occasionally by *mig*. Remote invocation is performed automatically by those applications, while migration is normally performed only when processes are evicted. A load-average daemon on each host is responsible for advertising the host's availability and for evicting foreign processes when the host is reclaimed.

[1]Note that there is no special support to evict processes when a host is used by someone not physically located with it. Someone who logs into a workstation remotely, or over a phone line, may have to compete with foreign processes for resources. We consciously chose to distinguish between physical and remote access, because users tend to notice any performance degradation of window-based interactive applications more than applications run remotely. Of course, the same mechanisms used to detect local interactive input could be extended to evict processes when a user interacts with a host remotely, if that were desired.

Chapter 4

Process Migration Mechanism

4.1 Introduction

In general, migrating a process involves two phases. The first phase consists of extracting the process state from one host (the source) and installing it on another host (the target). The second phase begins when the system starts executing the process on the target, and ends when the process terminates or migrates elsewhere. These two phases interact, because the actions the system performs during the first phase affect what special operations are necessary during the second.

The first phase, process transfer, depends on the state associated with a process. Section 4.2 examines different types of process state in a distributed system. It also examines ways to manage that state when a process changes hosts: transferring state, forwarding operations, or sacrificing transparency. Finally, it describes the step-by-step procedure used to migrate a process in Sprite.

The second phase, process execution, depends not only on the way in which state is transferred, but also the degree to which migration is intended to be transparent. For some operations, a system can trade off the simplicity of forwarding operations against the cost of inter-host communication; for others, the host on which a process executes is the only possible provider of a resource. In Section 4.3, I describe how Sprite uses the *home machine* to provide transparency for a remote process: the home machine manages the few kernel calls that are location-dependent, and it forwards operations on the process, such as signals, to the kernel on the remote host.

Section 4.4 discusses the relationship between process migration and residual dependencies, which require a process on one host to receive services from another host on which it previously executed. As I discussed above in Section 2.2, residual dependencies are generally bad for both reliability and performance, though some exceptions exist. Since in Sprite we place workstation autonomy above any considerations of migration performance, Sprite leaves no residual dependencies after eviction. However, Sprite does have dependencies on the home machine to support transparency.

Finally, Section 4.5 summarizes the chapter.

4.2 Process Transfer

The techniques used to migrate a process depend on the state associated with the process being migrated. If a stateless process existed, then migrating such a process would be trivial. In reality processes have large amounts of state, and both the amount and variety of state seem to be increasing as operating systems evolve. The more variety of state, the more complex the migration mechanism is likely to be. Process state typically includes the following:

- **Virtual memory**. In terms of size, the greatest amount of state associated with a process is likely to be the memory that it accesses. Thus the time to migrate a process is often limited by the speed of transferring virtual memory.

- **Open files**. If the process is manipulating files or devices, then there will be state associated with these open channels, both in the virtual memory of the process and also in the operating system kernel's memory. The state for an open file includes the internal identifier for the file, the current access position, and possibly cached file blocks. The cached file blocks may represent a substantial amount of storage, in some cases greater than the process's virtual memory.

- **Message channels**. In a message-based operating system such as Mach [ABB^{+}86] or V [Che88], state of this form would exist in place of open files. (In such a system message channels would be used to access files, whereas in Sprite, file-like channels are used for interprocess communication.) The state associated with a message channel includes buffered messages plus information about senders and receivers.

- **Execution state**. This consists of information that the kernel saves and restores during a context switch, such as register values and condition codes.

- **Other kernel state**. Operating systems typically store other data associated with a process, such as the process's identifier, a user identifier, a current working directory, signal masks and handlers, resource usage information, references to the process's parent and children, and so on.

For each portion of the state associated with a process, the system must do one of three things during migration: transfer the state, arrange for forwarding, or use comparable state on the target and sacrifice transparency. To transfer a piece of state, it must be extracted from its environment on the source machine, transmitted to the target machine, and reinstated in the process's new environment on that machine. For state that is private to the process, such as its execution state, transfer is relatively straightforward. Other state, such as kernel state distributed among complex data structures (possibly on multiple hosts), may be much more difficult to extract and reinstate. An example of "difficult" state in Sprite is information about open files, particularly those being accessed on remote file servers. Lastly, some state may be impossible to transfer. Such state is usually associated with physical devices on the source machine. For example, the frame buffer associated with a display must remain on the machine containing the display; if a process with access to the frame buffer migrates, it will not be possible to transfer the frame buffer.

The second option for each piece of state is to arrange for forwarding. Rather than transfer the state to stay with the process, the system may leave the state where it is and forward operations back and forth between the source and target machines. For example, I/O devices cannot be transferred, but the operating system can arrange for output requests to be passed back from the process to the device, and for input data to be forwarded from the device's machine to the process. In the case of message channels, arranging for forwarding might consist of changing sender and receiver addresses so that messages to and from the channel can find their way from and to the process. Ideally, forwarding should be implemented transparently, so that it is not obvious outside the operating system whether the state was transferred or forwarding was arranged.

The third option, sacrificing transparency, is a last resort: if neither state transfer nor forwarding is feasible, then one can ignore the state on the process's current and simply use the corresponding state on the target after migration. The only situation in Sprite where neither state transfer nor forwarding seemed reasonable is for memory-mapped I/O devices such as frame buffers, as alluded to above. We decided to disallow migration for processes using these devices.

In a few rare cases, lack of transparency may be desirable. For example, a process that requests the amount of physical memory available should obtain information about its current host rather than its home machine. For Sprite, a few special-purpose kernel calls (*e.g.*, calls that read instrumentation counters in the kernel) are also intentionally non-transparent with respect to migration. In general, though, it would be unfortunate if a process behaved differently after migration than before.

The subsections below describe how Sprite deals with the various components of process state during migration. The solution for each component consists of some combination of transferring state and arranging for forwarding.

4.2.1 Virtual Memory Transfer

Virtual memory transfer is the aspect of migration that has been discussed the most in the literature, perhaps because it is believed to be the limiting factor in the speed of migration [Zay87b]. One simple method for transferring virtual memory is to send the process's entire memory image to the target machine at migration time, as in Charlotte [AF89] and LOCUS [PW85]. This approach is simple but it has two disadvantages. First, the transfer can take many seconds, even using the highest transfer rate allowed by the network. During this time the process is *frozen*: it cannot execute on either the source or destination machine. For some processes, particularly interactive ones, long freeze times may be unacceptable. The second disadvantage of a monolithic virtual memory transfer is that it may result in wasted work for portions of the virtual memory that are not used by the process after it migrates. The extra work is particularly unfortunate (and costly) if it requires old pages to be read from secondary storage. For these reasons, several other approaches have been used to reduce the overhead of virtual memory transfer; the mechanisms are illustrated in Figure 4.1 and described in the paragraphs below.

In Section 2.3.3, I described techniques that two other systems, V and Accent, used to address the cost of copying memory. To summarize, the V System allows a process to continue executing on the source host while its address space is transferred to the target [The86, TLC85], while Accent transfers a process and retrieves its memory image as the memory is referenced [Zay87a, Zay87b]. In V, pre-copying reduces freeze times substantially, but the need to copy pages multiple times can increase the total amount of work to migrate a process. In Accent, lazy copying permits a process to begin execution on the target with minimal freeze time, and it reduces the cost of migration because pages that are not used are never copied at all. However, lazy copying leaves residual dependencies on the source machine: the source must store the unreferenced pages and provide them on demand to the target. In the worst case, a process that migrates several times could leave virtual memory dependencies on any or all of the hosts on which it ever executed.

Sprite's migration facility uses a different form of virtual memory transfer that takes advantage of our existing network services while providing some of the advantages of lazy copying. In Sprite, backing storage for virtual memory is implemented using ordinary files. Since these backing files are stored in the network file system, they are accessible throughout the network. During migration the source machine freezes the process, flushes its dirty pages to backing files, and discards its address space. On the target machine, the process starts executing with no resident pages and uses the standard paging mechanisms to load pages from the backing files as they are needed.

In most cases no disk operations are required to flush dirty pages in Sprite. The backing files are stored on network file servers, which cache recently-used file data in memory. When the source machine flushes a dirty page it is simply transferred over the network to the server's main-memory file cache. If the target machine accesses the page then it is retrieved from the cache. Disk operations will occur only if the server's cache overflows.

Sprite's virtual memory transfer mechanism was simple to implement because it uses pre-existing mechanisms both for flushing dirty pages on the source and for handling page faults on the target. It has some of the benefits of the Accent lazy-copying approach since only dirty pages incur overhead at migration time; other pages are sent to the target machine when they are referenced. The Sprite approach will require more total work than Accent's, though, since dirty pages may be transferred over the network twice: once to a file server during flushing, and once later to the destination machine.

The Sprite approach to virtual memory transfer fits well with the way migration is

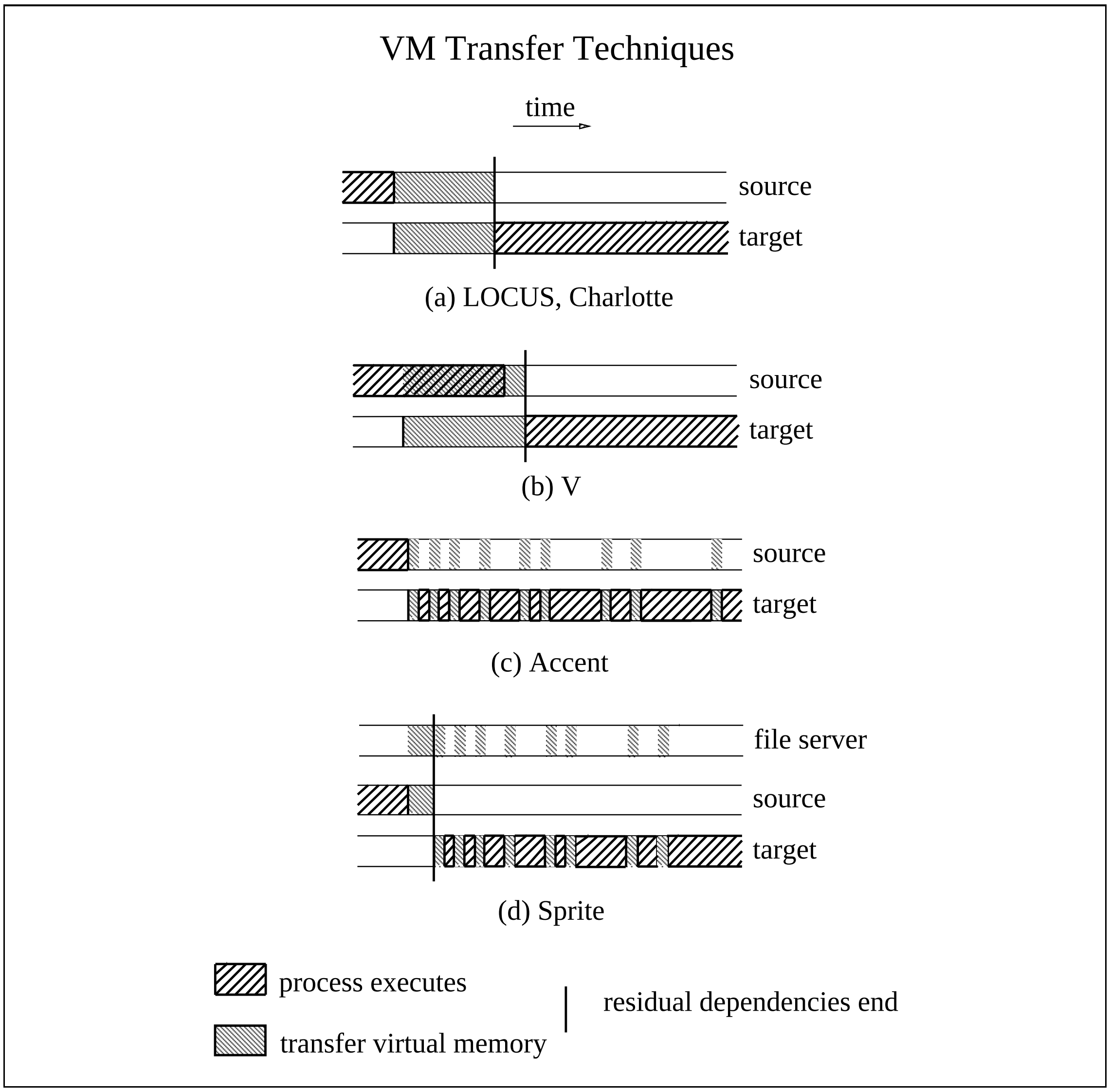

Figure 4.1: *Different techniques for transferring virtual memory.* (a) shows the scheme used in LOCUS and Charlotte, where the entire address space is copied at the time a process migrates. (b) shows the pre-copying scheme used in V, where the virtual memory is transferred during migration but the process continues to execute during most of the transfer. (c) shows Accent's lazy-copying approach, where pages are retrieved from the source machine as they are referenced on the target. Residual dependencies in Accent can last for the life of the migrated process. (d) shows Sprite's approach, where dirty pages are flushed to a file server during migration and the target retrieves pages from the file server as they are referenced. In the case of eviction, there are no residual dependencies on the source after migration. When a process migrates away from its home machine, it has residual dependencies on its home throughout its lifetime.

typically used in Sprite. Process migration occurs most often during an *exec* system call, which completely replaces the process's address space. If migration occurs during an *exec*, the new address space is created on the destination machine so there is no virtual memory to transfer. As others have observed (*e.g.*, LOCUS [PW85]), the performance of virtual memory transfer for *exec*-time migration is not an issue. Virtual memory transfer *is* an issue, however, when migration is used to evict a process from a machine whose user has returned. In this situation the most important consideration is to remove the process from its source machine quickly, in order to minimize any performance degradation for the returning user. Sprite's approach works well in this regard since (a) it does the least possible work to free up the source's memory, and (b) the source need not retain pages or respond to later paging requests as in Accent. It would have been more efficient overall to transfer the dirty pages directly to the target machine instead of a file server, but this approach would have added complexity to the migration mechanism.

Virtual memory transfer becomes much more complicated if the process to be migrated is sharing writable virtual memory with some other process on the source machine. In principle, it is possible to maintain the shared virtual memory even after one of the sharing processes migrates [LH89], but this changes the cost of shared accesses so dramatically that it seemed unreasonable. Currently, shared writable virtual memory almost never occurs in Sprite, so Sprite simply disallows migration for processes using it. A better long-term solution would be to migrate all the sharing processes together, but even this may be impractical if there are complex patterns of sharing that involve many processes.

4.2.2 Migrating Open Files

When a process migrates, it must have uninterrupted access to any files it has opened. Either the state associated with each file must be transferred to the target, or operations on the file must be forwarded to the machine on which the file was opened. As I shall demonstrate in Chapter 5, the migration mechanism would be much simpler with the "arrange for forwarding" approach for open files than with the "transfer state" approach. However, the frequency of file-related kernel calls and the cost of forwarding a kernel call over the network make the forwarding approach unacceptable. Forwarding file-related calls would slow down the remote process and load the machine that stores the file state. Sprite workstations are typically diskless and files are accessed remotely from file servers, so the forwarding approach would cause each file request to be sent over the network once to the machine where the file was opened, and probably a second time to the server. From a performance standpoint, it would be desirable to transfer open-file state along with a migrating process and then use the normal mechanisms to access the file (*i.e.*, communicate directly with the file's server).

Because the performance considerations outweighed any concerns about increased complexity, Sprite uses the "transfer state" approach for open files. Once an open file has been transferred to a new host, access to the file is managed using standard mechanisms in the Sprite file system. The server that stores a file (or controls a device) is responsible for keeping two things consistent: the file's contents, and processes' offsets into the file. If processes on different hosts access the same file simultaneously, and a process has the file open for writing, the Sprite cache consistency protocol causes the file to become uncacheable on all hosts except the server that stores the file. This policy prevents hosts from storing inconsistent images of the file in their caches. Also, if migration causes processes on different hosts to share a single file descriptor, the offset into the file is managed by the file's server, and all operations using the file descriptor are forwarded to the server.

Transferring open-file state provides high performance at the cost of considerable increased complexity; in fact, bookkeeping between file servers and migrating processes on client workstations is the most complicated aspect of the Sprite process migration facility. Locking and updating the data structures for an open file simultaneously on multiple hosts provides numerous opportunities for deadlocks, race conditions, and inconsistent reference counts. The next chapter discusses in depth the interaction between process migration and the file system.

4.2.3 The Process Control Block

Aside from virtual memory and open files, the main remaining issue is how to deal with the process control block (PCB) for the migrating process: should it be left on the source machine or transferred with the migrating process? In Sprite I use a combination of both approaches. The home machine for a process must assist in some operations on the process (see Section 4.3 for details), so it always maintains a PCB for the process. In addition, the current machine for a process also has a PCB for it. If a process is migrated, then most of the information about the process is kept in the PCB on its current machine; the PCB on the home machine serves primarily to locate the process and most of its fields are unused.

The other elements of process state besides virtual memory and open files are much easier to transfer than virtual memory and open files, since they are not as "bulky" as virtual memory and they don't involve distributed state like open files. At present the other state consists almost entirely of fields from the process control block, such as signal masks, process groups, and identifiers. In general, all that needs to be done is to transfer these fields to the target machine and reinstate them in the process control block on the target.

4.2.4 The Fragility Problem

One problem with process migration is that it involves state that is manipulated by virtually every major module in the kernel. This makes it hard to separate the implementation of migration from the implementation of the other kernel modules, since changes in one can affect the other. As a result, migration has been one of the most fragile parts of the Sprite kernel. It often breaks when seemingly unrelated parts of the kernel are modified. I believe that the problem is inherent in the nature of migration (Theimer had similar problems in his implementation [The86]), but I have used two techniques to lessen the difficulties.

My first approach to the problem of migration fragility was to introduce *migration version numbers*. Before Sprite started using migration version numbers it was very difficult to make any changes to the migration mechanism, or to kernel data structures affected by migration. As the kernel changed, some machines would be running new kernels and some would run older kernels, without the changes. When an attempt was made to migrate between new and old kernels, the result was inevitably a crash of one or both machines. The only way to prevent these crashes was to reboot every Sprite host whenever anyone made an incompatible change. To catch incompatibilities, Sprite kernels now include a "migration version number," which is incremented whenever any aspect of the migration protocol changes. Migration is disallowed between machines with different version numbers, so that incompatible versions can coexist in the same network.

My second approach was to avoid centralizing the migration code in one place. Instead, I have distributed it among the various kernel modules whose state is affected. By placing the migration-related code next to the other code that manipulates state information, I have found it easier to keep the two consistent. For each piece of state that must be considered during migration there exist four procedures that are invoked during migration (see Section 4.2.5 for details). All that is needed to deal with additional state during migration is to write the four procedures for that state and enter their names into a table used during migration. At present the table contains entries for the following pieces of state: basic machine-independent process state, machine-dependent execution state, virtual memory, files, signals, profiling, and arguments to *exec*.

4.2.5 Migration Procedure

The overall algorithm for migrating a process from a source machine to a target machine is table-driven and involves the following steps:

1. The process is signaled to make it trap into the kernel. (At the point when the signal is handled, the state of the process within the kernel is simple and well-defined, since the process is not in the midst of a kernel call.)

2. If the process is not being migrated back to its home machine, the source kernel contacts the target to confirm that it is running, that it is available for migration, and that it has the same migration version number as the source. The kernel on the target host allocates a new process control block and returns a token that is later used to identify the new instance of the process on the target. If the process is migrating home, then the three conditions are all known to be satisfied, and no preliminary communication is necessary. In that case, the token for the process is simply its process identifier on the home machine.

3. For each module with process state that must be transferred to the new host, the source kernel calls a *pre-migration routine* to obtain the size of the encapsulated data. This routine may, as a side effect, initiate any actions required to migrate state:

for example, the virtual memory pre-migration routine queues the process's modified pages for flushing.

4. The source kernel allocates a buffer to hold the combined encapsulated state. This buffer varies in size primarily as a function of the size of the process's page tables; for a small process, a typical size for this buffer would be about 5 Kbytes.

5. For each module, the source kernel calls an *encapsulation routine* to place that module's state into a portion of the buffer. As with the pre-migration routines, encapsulation routines may have side effects, such as waiting for modified pages to be flushed or communicating with file servers for open files.

6. The source kernel passes the encapsulated state to the target kernel via a single remote procedure call. On the target, the corresponding *de-encapsulation routine* is invoked for each portion of the buffer.

7. For each module, if a *post-migration routine* has been specified, the kernel on the source host invokes the procedure to clean up any remaining state.

8. The source kernel frees the buffer and informs the target to resume the process.

4.3 Supporting Transparency: Home Machines

As I discussed in Section 1.1, transparency was one of my most important goals in implementing migration. By "transparency" I mean two things in particular. First, a process's behavior should not be affected by migration. Its execution environment should appear the same, it should have the same access to system resources such as files and devices, and it should produce exactly the same results as if it hadn't migrated. Second, a process's appearance to the rest of the world should not be affected by migration. To the rest of the world the process should appear as if it never left its original machine, and any operation that is possible on an unmigrated process (such as stopping, debugging, or signalling) should be possible on a migrated process.

Both of these two aspects of transparency are defined with respect to a process's *home machine*, which is the machine where it would execute if there were no migration at all. Even after migration, everything should appear as if the process were still executing on its home machine. As will be seen below, Sprite achieves transparency by involving the home machine in some operations for remote processes.

4.3.1 Messages Versus Kernel Calls

On the surface, it might appear that transparency is particularly easy to achieve in a message-based system like Accent [Zay87b], Charlotte [AF89], or V [Che88]. In these systems all of a process's interactions with the rest of the world occur in a uniform fashion through message channels. All that is needed to guarantee transparency is to preserve the behavior of the message channels by forwarding messages to and from a remote process's new location. This is typically done by updating addresses in the endpoints of the message channels. For example, Charlotte updates the addresses at the time of migration, while V waits until the old incorrect address is used and then updates the address using a multicast protocol.

In contrast, transparency might seem harder to achieve in a system like Sprite that is based on kernel calls. In such a system the state of the process is expected to be in the kernel of the machine where the process executes. This requires that the state be transferred during migration, which is more complicated than forwarding.

It turns out that neither of these initial impressions is correct. For example, it would be possible to implement forwarding in a kernel-call-based system by leaving *all* of the kernel state on the home machine and using remote procedure calls to forward home every kernel call, as Remote UNIX [Lit87] does. This would result in an approach very similar to forwarding messages, and our initial plan was to use an approach like this for Sprite.

Unfortunately, an approach based entirely on forwarding kernel calls or forwarding messages will not work in practice, for two reasons. The first problem is that some services must necessarily be provided on the machine where a process is executing. If a process invokes a kernel call to allocate virtual memory (or if it sends a message to a memory server to allocate virtual memory), the request *must* be processed by the kernel or server on the machine where the process executes, since only that kernel or server has control over the machine's page tables. Forwarding is not a viable option for such machine-specific functions: state for these operations must be migrated with processes. The second problem with forwarding is cost. It will often be much more expensive to forward an operation to some other machine than to process it locally. If a service is available locally on a remote process's new machine, it will be more efficient to use the local service than to forward operations back to the service on the process's old machine. Furthermore, forwarded calls increase the load on the old machine, so the less forwarding is needed the more parallelism is possible.

4.3.2 Achieving Transparency

Transparency is achieved in practice by combining several approaches. I used four different techniques in Sprite. The most desirable approach is to make kernel calls location-independent; Sprite has been gradually evolving in this direction. For example, in early versions of the system we permitted different machines to have different views of the file system name space. This improved the generality of the system but required *open* and several other kernel calls to be forwarded home after migration, imposing about a 20% penalty on the performance of remote compilations. In order to simplify migration (and for several other good reasons), we changed the file system so that every machine in the network sees the same name space. This made the *open* kernel call location-independent, so no extra effort was necessary to make *open* work transparently for remote processes.

Another example of the evolution toward transparency is the *kill* kernel call, which sends a signal to a process. Sprite has a single network-wide name space for process identifiers, and we made *kill* work regardless of whether the process being signalled was on the same machine as the process issuing the kernel call; this allowed *kill* to be used transparently by remote processes.

The second technique is to transfer state from the source machine to the target at migration time, so that normal kernel calls may be used after migration. For example, a process's virtual memory is transferred at migration time so that the kernel's virtual-memory-related state for a remote process is identical to an unmigrated process's state. This allows Sprite to use the normal virtual-memory-related kernel calls for remote processes without any loss of transparency. Sprite also uses the state-transfer approach for open files, process and user identifiers, resource usage statistics, and a variety of other things.

Note that from an implementation standpoint, the first two techniques are quite similar. In either case, a kernel call is handled entirely by the kernel of the host on which a process executes, without any special considerations due to the process being associated with a different host. The techniques differ only to the extent that state must be transferred with a process to enable calls to be handled on the remote host. In fact, the two techniques overlap to some extent: the *open* kernel call is location-independent, but in order to allow the remote host to handle the *open* call without forwarding it home, the process's current working directory must be transferred at migration time.

The third technique is to forward kernel calls home. In almost all cases, forwarding calls can be done without any special interpretation by the remote kernel. Instead, the kernel encapsulates the arguments to the kernel call, transfers the arguments to the process's home machine with an indication of which call is to be performed, and returns the results to the user process. This technique was originally used for a large number of kernel calls, but for reasons of performance and complexity I have gradually replaced most uses of forwarding with transparency or state transfer. At present there are only a few kernel calls that cannot be implemented transparently and for which we cannot easily transfer state. The most important such kernel call is *gettimeofday*, which returns the current time. Clocks are not synchronized between Sprite machines, so for remote processes Sprite forwards the *gettimeofday* kernel call back to the home machine. This guarantees that time advances monotonically even for remote processes, but incurs a performance penalty for processes that read the time frequently. Despite the performance penalty, however, forwarding calls is a simple solution to a wide range of problems, and several other systems forward calls automatically in some or all cases. For example, Plan 9 [PPTT90] forwards *gettimeofday* calls from processor servers to a user's own machine in order to keep times consistent across hosts, and MOSIX [BS85] and Remote UNIX [Lit87] forward calls as described in Section 2.3.

Forwarding also occurs from the home machine to a remote process's current machine. For example, when a process is signalled (*i.e.*, when some other process specifies its identifier in the *kill* kernel call), the signal operation is sent initially to the process's home machine.

If the process is not executing on the home machine, then the home machine forwards the operation on to the process's current machine. The performance of such operations could be optimized by retaining a cache on each machine of recently used process identifiers and their last known execution sites. This approach is used in LOCUS and V and allows many operations to be sent directly to a remote process without passing through another host. An incorrect execution site is detected the next time it is used and correct information is found by sending a message to the host on which the process was created (LOCUS) or by multicasting (V).

The fourth "approach" is really just a set of *ad hoc* techniques for a few kernel calls that must update state on both a process's current execution site and its home machine. One example of such a kernel call is *fork*, which creates a new process. Process identifiers in Sprite consist of a home machine identifier and an index of a process within that machine. Management of process identifiers, including allocation and deallocation, is the responsibility of the home machines named in the identifiers. If a remote process *fork*s, the child process must have the same home machine as the parent, which requires that the home machine allocate the new process identifier. Furthermore, the home machine must initialize its own copy of the process control block for the process, as described in Section 4.2.3. Thus, even though the child process will execute remotely on the same machine as its parent, both its current machine and its home machine must update state. The flow of control is essentially identical to a *fork* call by a local process, with an additional RPC at one point to inform the home machine about the child process.

Similar kinds of cooperation occur for *exit*, which is invoked by a process to terminate itself, and *wait*, which is used by a parent to wait for one of its children to terminate. The *wait* call is a special case, because it involves both state retrieval and state update. When a remote process waits for a child to exit, the kernel on the remote host contacts the process's home machine to obtain information about an exiting child; if no child has exited, then the kernel on the home machine notes the parent's interest and contacts the remote host when a child exits later. Using the home machine to maintain the state for the *wait-exit* rendezvous point has permitted Sprite to avoid several potential race conditions between a process exiting, its parent waiting for it to exit, and one or both processes migrating. LOCUS similarly uses the site on which a process is created to synchronize operations on the process.

The disposition of each kernel call is listed in Appendix A on page 109. Currently, 11 out of 106 system calls are encapsulated and "forwarded blindly" to the home machine, and 4 calls are performed via cooperation between the kernels of both machines. The remaining 91 calls are handled by the remote host using the normal system call interface, with no special actions by the home machine. Figure 4.2 depicts the alternative paths that kernel calls by foreign processes can follow.

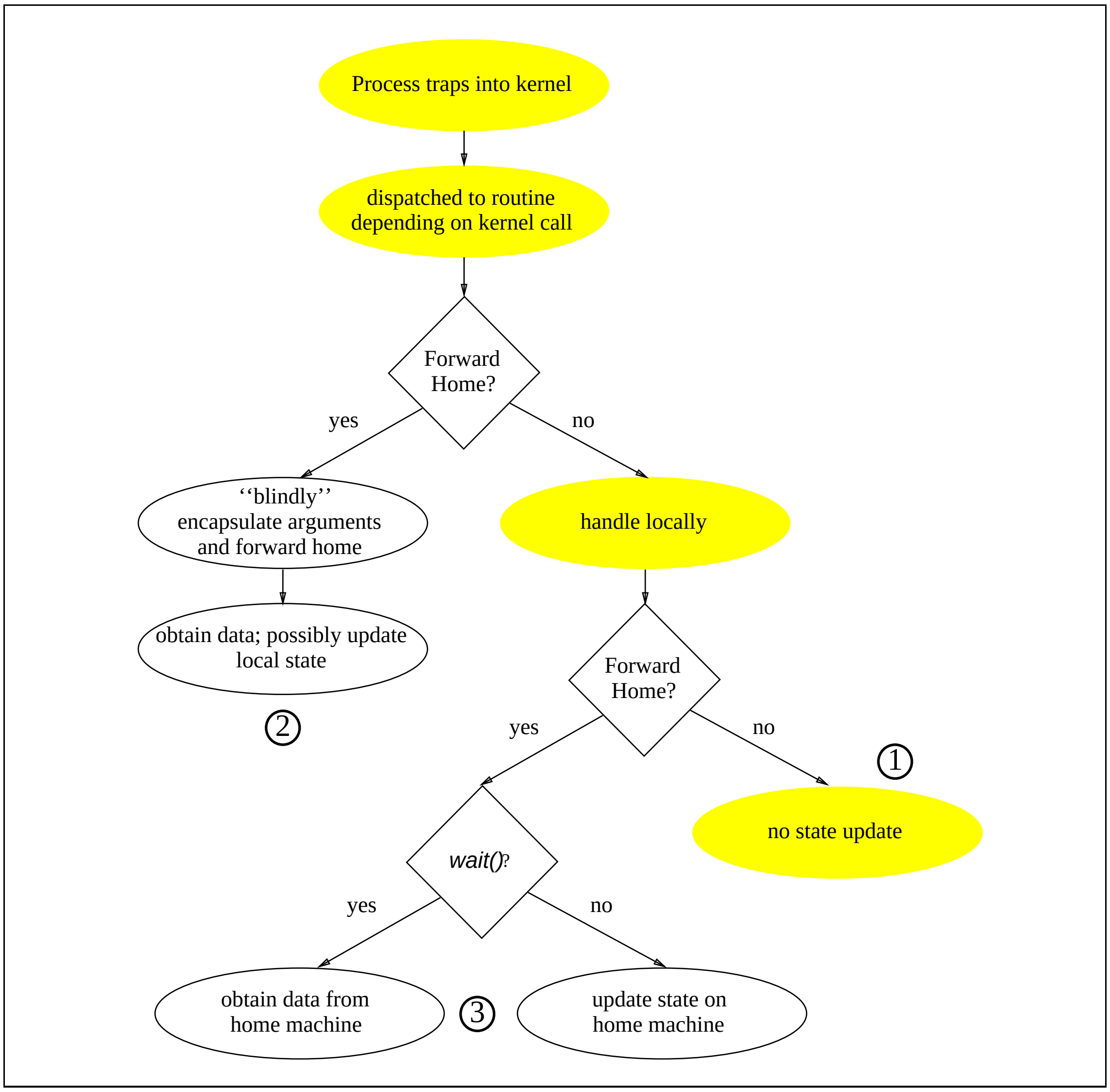

Figure 4.2: *Transparent management of kernel calls by a foreign process.* When a process calls into the kernel on a host that is not its home machine, the kernel handles the call in one of three ways. (1) Most commonly, kernel calls are handled locally without any need to involve the home machine. These calls follow the path indicated by the shaded ovals. (2) In some cases, such as *gettimeofday*, calls are forwarded home with no interpretation by the remote host. (3) Finally, in a few cases such as *fork* and *wait*, calls are handled locally initially but at some point involve the home machine to update state or obtain data.

4.4 Residual Dependencies

I have defined a *residual dependency* as a dependency of a process on a host after the process migrates away from the host. One example of a residual dependency occurs in Accent, where a process's virtual memory pages are left on the source machine until they are referenced on the target. Another example occurs in Sprite, where the home machine must participate whenever a remote process forks or exits.

Despite the disadvantages of residual dependencies, mentioned previously, it may be impractical to eliminate them all. In some cases dependencies are inherent, such as when a process is using a device on a specific host; these dependencies cannot be eliminated without changing the behavior of the process. In other cases, dependencies are necessary or convenient to maintain transparency: for example, in Sprite the home machine serves as a rendezvous point for information about exiting processes that may have executed on different hosts. Lastly, residual dependencies may actually improve performance in some cases, such as lazy copying in Accent, by deferring state transfer until it is absolutely necessary.

With respect to residual dependencies, Sprite distinguishes between dependencies on the home machine and dependencies after eviction. Because I believed transparency would require certain dependencies on the home machine (for example, migrated processes appearing in a list of active processes on a user's own machine), I permitted some residual dependencies on the home machine where those dependencies made it easier to implement transparency. As described in Section 4.3.2 above, there are only a few situations where the home machine must participate, so the performance impact is minimal (see Chapter 7 for measurements). However, these dependencies prevent users from migrating processes in order to survive the failure of their home machine. In Chapter 9 I consider a nontransparent variant of process migration, which would change the home machine of a process when it migrates and break all dependencies on its previous host. Nontransparent process migration might be appropriate in cases where reliability is more important than transparency. For example, long-running noninteractive applications might make no reference to their location or the time of day, and the failure of the home machine would cause significant amounts of processing to be lost if a remote process were terminated.

Although Sprite permits residual dependencies on the home machine, it does not leave dependencies on any other machines. If a process migrates to a machine and is then evicted or migrates away for any other reason, there will be no residual dependencies on that machine. This provides yet another assurance that process migration will not impact users' response when they return to their workstations. The only noticeable long-term effect of foreign processes is the resources they may have utilized during their execution: in particular, the user's virtual memory working set may have to be demand-paged back into memory upon the user's return.

4.5 Summary

When a process migrates, its execution environment must continue to be accessible to it. Much of its state must reside on the host on which the process executes, while other pieces of its environment can be accessed from its former host. For each piece of state associated with a process, the system can move the state with the process at the time of migration, or arrange to forward the state when it is needed.

If state must be forwarded, then the process has a residual dependency on the host that performs the forwarding. Such a dependency may increase the likelihood of process failure: if the host with the dependency fails, then the process may be unable to access resources. The dependency also may degrade the performance of the host providing the resource—if, for example, the resource is virtual memory. On the other hand, some dependencies are unavoidable if migration is to be transparent, and the same dependencies that degrade performance during execution can reduce the time needed to migrate a process.

Implementing migration is complicated, and insulating migration from the rest of the kernel is especially difficult. The more modular migration is, the easier it is to make process migration work in the presence of changes elsewhere in the system. In Sprite, the interaction between the file system and process migration is the most complex aspect of migration. The next chapter discusses this interaction, in the context of distributed systems in general and Sprite in particular.

Chapter 5

Interaction with the File System

5.1 Introduction

This chapter describes the relationship between process migration and the Sprite file system. Though the file system simplifies migration by providing a single system-wide shared name space and transparent access to files and devices on different hosts, it also has proven to be the most difficult aspect of migration to manage. The complexity arises from two sources: UNIX semantics, which complicates the process migration mechanism, and file data caching, which affects the performance of migration.

With respect to process migration, the most demanding requirements of UNIX compatibility are shared file access positions and references to deleted files. First, in a UNIX-like system such as Sprite, processes can share a common access position for a file as a result of inheriting files through process creation. In a system without process migration, access positions can be shared only by processes on a single machine, so they do not create a problem. Unfortunately, migration can cause processes on different machines to share a common stream and therefore share a single access position into a file. (Figure 5.1 depicts a possible scenario.) Keeping the shared access position consistent in the face of multiple migrations and process creations can be difficult, because different hosts may update the access position simultaneously. Second, a file cannot be deleted if a process has it open; instead, the file is deleted when the last reference to it is removed. When an open file migrates, the reference to the file must never appear to be removed or the file might inadvertently be deleted.

File data caching is the other principal source of complexity for migration. The Sprite file system provides high performance by permitting hosts to use nearly all of memory to cache file data. File servers are responsible for guaranteeing "consistent access" to the cached data; this is performed by disabling caching when files are accessed by multiple hosts while any host is writing the file, and by flushing modified data to file servers if a process on a different host opens a file [NWO88]. Sprite's caching policy impacts process migration and load sharing in two ways:

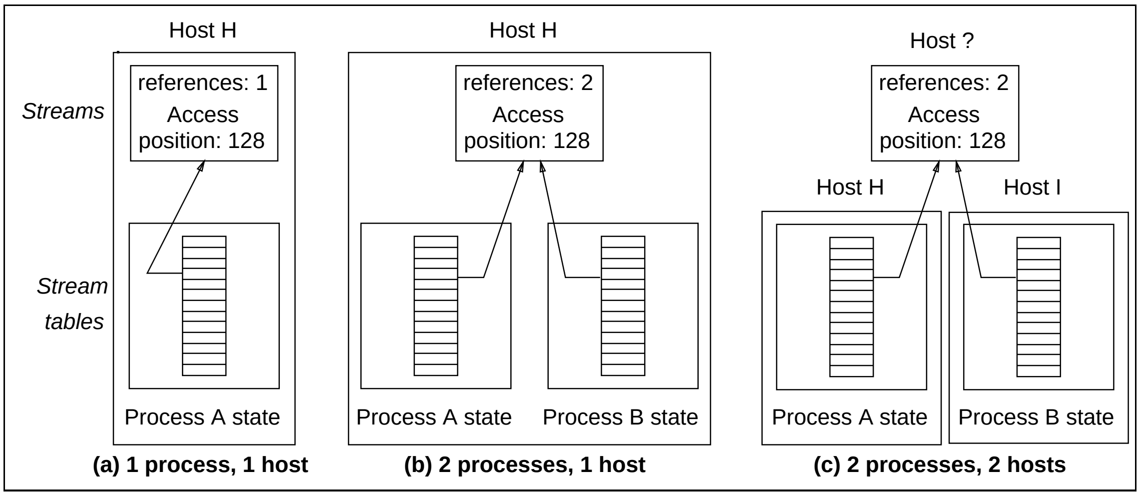

Figure 5.1: *Shared streams across a network.* In (a), process A on host H contains a reference to a particular stream, managed by H. (b) After the process forks a new child, B, the stream is shared by two processes on H, and is still managed by H. (c) If process B migrates to host I, the stream is shared by processes on host H and host I. Managing a stream that is shared by processes on different hosts requires a mechanism for synchronizing use of the access position. The access position may be managed by H, but as I show in Section 5.3.2, it may simplify matters to manage it on another host.

Process transfer. Migrating a process may cause a cacheable file to become uncacheable, or vice-versa, because the set of hosts accessing the file changes. The implementation of the file system is therefore complicated by the need to manage each of several possible cases. Also, modified data must be flushed to file servers before the migration can complete, thereby slowing down migration.

Load sharing. Data must also be flushed when a process on one host accesses the output of a process from another host. Without load sharing, consistency actions of this type would be infrequent, but load sharing can cause many hosts to write data that a single host will later read. (For example, *pmake* could execute compilations on several different hosts, then run the linker on one of the hosts. The object files read by the linker would be flushed to their server before the linker could access them.) Note that cache flushes due to load sharing do not affect complexity, but the time needed to obtain the data from each host can reduce the benefits of executing tasks in parallel.

The rest of this chapter is organized as follows. The next section discusses the Sprite file system in detail and defines some terminology that is used throughout this chapter. It describes the state associated with an open file and indicates how that state affects migration. Section 5.3 presents the mechanism used in Sprite to transfer open files and to support shared file access positions. Section 5.4 discusses the effect of file caching on the time to migrate processes and the execution speed of processes that use load sharing. Section 5.5 summarizes the chapter.

5.2 The Sprite File System

In order to discuss the relationship between migration and the file system in greater detail, it is necessary to describe the internal structure of the file system. Section 5.2.1 explains how Sprite provides transparent access to files and devices across a network, and it defines some terminology. Section 5.2.2 discusses cache consistency in Sprite.

5.2.1 Transparent File Access

The Sprite file system implements a single system-wide shared name space, and any Sprite host can access a file or device on any other host. This degree of transparency is obtained by separating the host that is responsible for *naming* an object from the host that controls *access* to the object. The host that names an object is known by the generic term of *file server*, while the host that controls its access is known as the object's *I/O server*. (Note that in the discussion in Section 5.1, the term "file server" usually referred to a host that is in fact an "I/O server.") A host that uses a file or device is known as a *client* of the I/O server.

The protocol for accessing a file system object depends on the type and location of the object. For ordinary files, the file server and I/O server are the same; to open one,

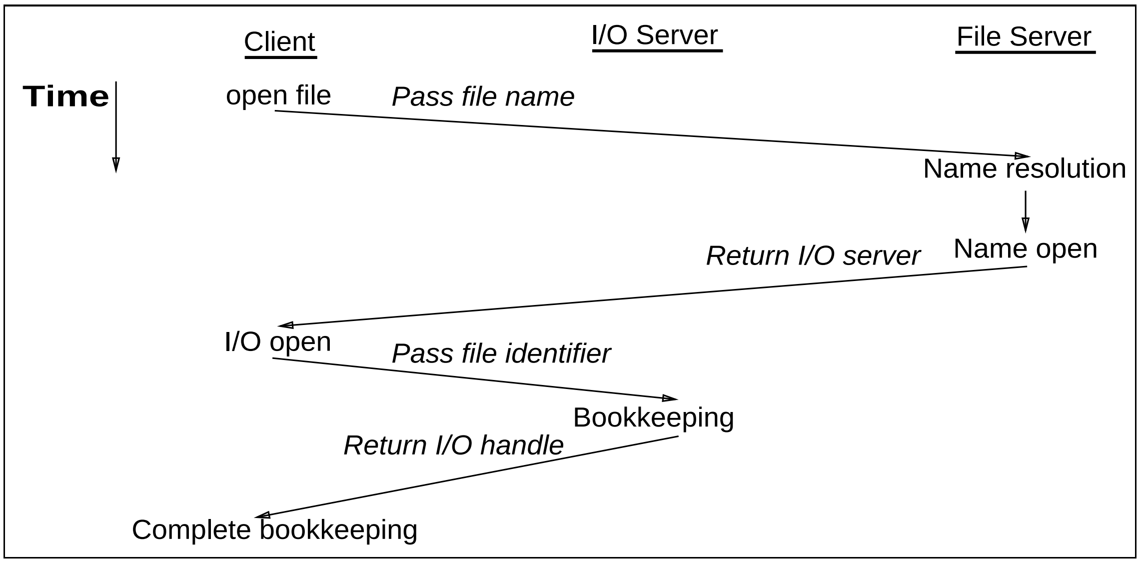

Figure 5.2: *File servers versus I/O servers.* When a process performs the *open* kernel call to open a file on a client workstation, the file server that manages the name for the file resolves the object referred to by the name and performs bookkeeping associated with the name. It returns the identity of the I/O server for the object, and the client contacts the I/O server to complete the operation of opening the file. The I/O server does its own bookkeeping to manage the file.

---

a client contacts the file server for the file, and it immediately obtains a token it uses to access the file in the future. For devices, however, the I/O server is often different from the server responsible for naming the device. Multiple communications are required to open an object with a different I/O server from file server. As an example, the name **/hosts/larceny/dev/printer** might refer to a device on the host named "larceny." To open the device, a process would perform a system call, which would involve an RPC to the file server responsible for the name "**/hosts/larceny/dev/printer**." That RPC would return a reference for the device, including an indication that the I/O server for the device is the host "larceny". Subsequent *read* or *write* operations, as well as some others, would be handled via RPC's to larceny. The relationship between file servers and I/O servers is depicted in Figure 5.2.

The state associated with an open file is managed hierarchically. Each process has a table of descriptors for open files, which map from integers to *I/O streams*. A stream includes a count of the processes using it, an access position into the file, and a pointer to a lower-level object known as an *I/O handle*. In turn, an I/O handle includes a count of the streams referring to it, the mode in which the file is accessed (reading, writing, and so on), and an identifier for a particular file on a particular I/O server [Wel90]. This hierarchy is depicted in Figure 5.3.

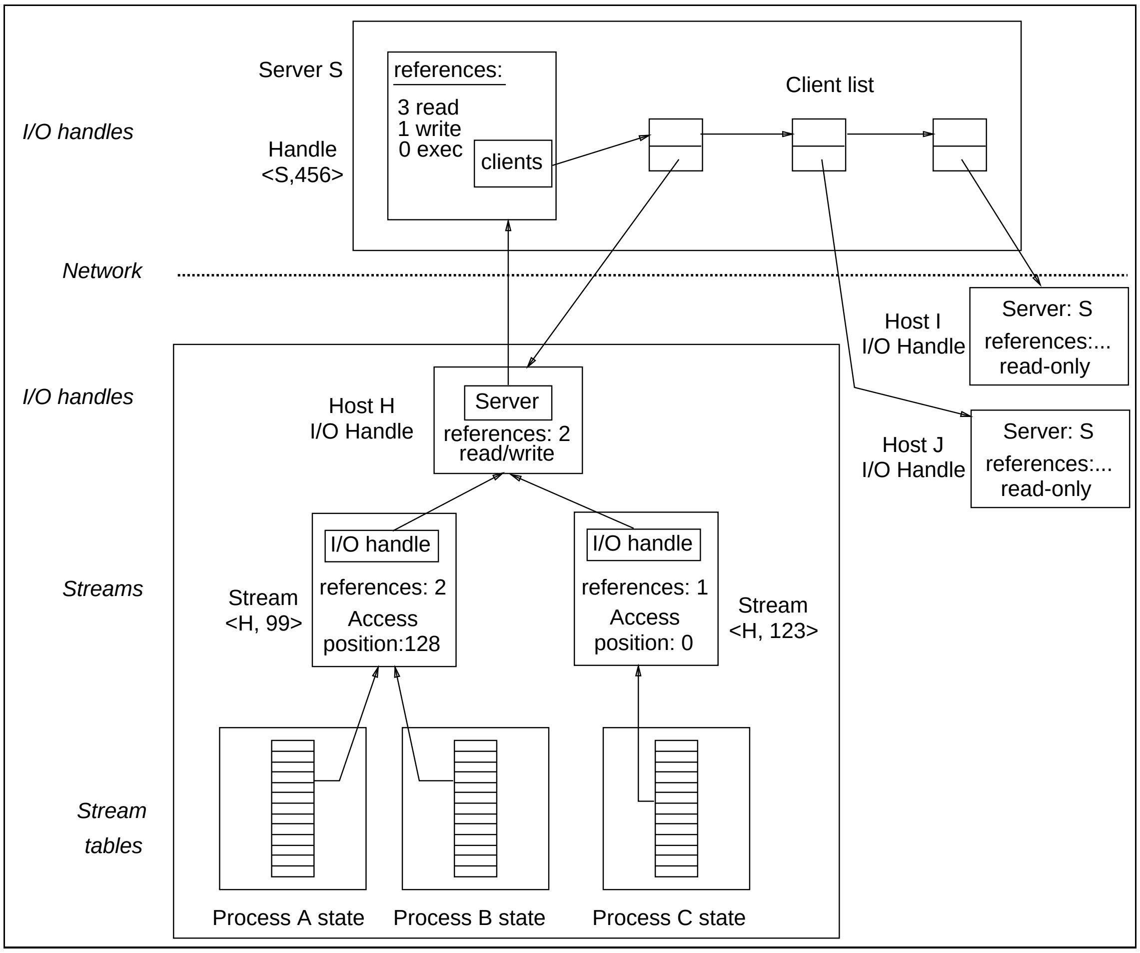

Figure 5.3: *File state.* Processes refer to streams, which refer to I/O handles. The I/O handles point to the I/O server, which stores its own copy of the I/O handle. The server keeps track of the usage of the file to guarantee consistent caching and to know when the last reference to a file is removed. In this example, host H has three processes using a total of two streams into a file identified as `<S,456>`. Hosts I and J also have streams that refer to handle `<S,456>`. The reference count shows that three hosts have readable streams for `<S,456>` and only one host has a writable stream to it.

5.2.2 File Data Caching

The I/O server for a file is responsible for ensuring consistent access to the file in the event that the file is shared by multiple hosts. The server keeps track of which hosts have the file open for reading and writing. If a file is open on more than one host and at least one of them is writing it, then caching is disabled; all hosts must forward their read and write requests for that file to the server so they can be serialized. With respect to file cacheability after migrating an I/O stream, there are four possible scenarios:

1. The file's status does not change. If no host is accessing the file in a writable mode, then it is cacheable on every host. Alternatively, if the file was already being accessed by multiple hosts with at least one host writing the file, it was uncacheable prior to migration and may continue to be uncacheable afterwards.

2. The file changes from uncacheable to cacheable. If the file is originally open on both the source and the target, and migration removes the last reference to the file on the source, then the file can be cached on the target.

3. The file changes from cacheable to uncacheable. This transition occurs if the file is open for writing on the source (and is not accessed on any other host), and then after migration is open on both the source and the target.

4. The file changes from being cacheable only on the source to being cacheable only on the target. As in the previous case, if the only use of the file is originally on the source, the file will be cacheable on the source, even if it is open for writing. If all streams that refer to the file move from the source to the target, the file will then be cacheable on the target. This case is similar to case 1; however, the file is cached on exactly one host at a time.

A table of the relative frequency of each case in practice appears in Chapter 8 on page 98.

When a writable open file is transferred during migration, if the file is cacheable on the source machine (as in cases 3 and 4 above), the file cache on the source may contain modified blocks for the file. These blocks are flushed to the file's I/O server during migration, so that after migration the target machine can retrieve the blocks from the file server without involving the source. This approach is similar to the mechanism for virtual memory transfer and thus has the same advantages and disadvantages. It is also similar to what happens in Sprite for shared file access without migration: if a file is opened, modified, and closed on one machine, then opened on another machine, the modified blocks are flushed from the first machine's cache to the server at the time of the second open. Section 5.4 discusses the performance implications of this approach to caching.

5.3 Transferring Open Files

The mechanism to transfer open-file state during migration has been implemented in Sprite twice, first by Mike Nelson and then by Brent Welch. In each case I assisted in debugging, but not in the overall design. Therefore, for the rest of the chapter, I use the term *we* when discussing work performed by multiple members of the Sprite research group, and *I* to refer to work I did on my own. (The same actually holds true elsewhere in this thesis but is much less pronounced.)

5.3.1 Close-and-reopen

In the first implementation, open files were transferred by encapsulating their state into a buffer, closing the files on the source, and reopening the files on the target. While this scheme worked in the majority of cases, it suffered from two deficiencies. First, since the file would actually be closed temporarily in the process of migrating the stream, it could be deleted completely if its reference count had become zero. Second, if processes shared a single stream and migration caused them to execute on different hosts, the stream would no longer be shared: by opening the file anew, the migrated process would have its own distinct access position for the file.

In practice, the problem of deleted files disappearing was more important than supporting shared access positions, because many UNIX utilities open files and immediately delete them so they will be removed upon termination. Therefore, we had to change file transfer to move the open-file reference from source to target without ever closing the file. We accomplished this by opening the file on the target before closing the file on the source. However, for writable files, this change had a different unexpected side-effect: since there was a window of time during which both the source and target had a file open, the file would appear to be write-shared and the server would disable caching for the file. Once the Sprite file system disables caching for a file, it does not reenable caching until all streams to the file have been closed, even if at some point only one host accesses the file. A transient write-sharing conflict would therefore disable caching indefinitely, adversely affecting performance.

It would have been possible to change Sprite's caching algorithm to avoid permanently disabling caching for a file, but that alone would not have permitted migrated processes to share file access positions. The next subsection describes an alternative approach.

5.3.2 Shared Access Positions with Atomic Transfer

Welch reimplemented the file system for several reasons, including the problems we had with process migration. He addressed the problem of shared access positions, and he implemented special code to support migrating streams atomically, so that the state maintained by the I/O server for crash recovery and cacheability could be kept consistent. In his thesis [Wel90], Welch describes the current implementation of the Sprite file system and its support for migration. In the remainder of this section, I highlight the important aspects of this implementation as it relates to migration.

Shared Access Positions

Just as Sprite permits only a single host to cache a writable file, it permits only a single host to manage the access position of a stream. With file caching, a client workstation is permitted to cache a writable file until it becomes shared, at which point only the I/O server for the file may cache it. With streams, a client workstation manages a stream used by its own processes until the stream becomes shared across the network, at which point only the I/O server for the stream manages the stream's access position.

To support this model of stream access, Welch separated the notions of *stream descriptors* and *stream references*. A stream *descriptor* is state associated with an open file, such as the access position; each stream descriptor is unique throughout the network. A stream *reference* associates a stream descriptor with a process, and as a result of migration, more than one host can have a reference to the same stream descriptor. Stream references can migrate between hosts, while a stream descriptor is stored by a single host.

Stream descriptors are used in one of two modes, depending on whether the stream is shared by multiple hosts. Normally, a stream is used by only a single host, and that host manages the stream descriptor (and therefore the access position). The I/O server for the file referenced by the stream stores a copy of the stream, containing information about the file it references and a unique identifier for the stream, but the I/O server's copy is not used. (Welch refers to the I/O server's copy of the stream descriptor as a *shadow stream*.) If a stream later becomes shared between machines, then each machine has a reference to the stream descriptor, which the file's I/O server manages. None of the hosts that use the stream stores the access position, nor do they cache the file. Instead, all operations on the file are forwarded to the server. (Thus, in Figure 5.1, the host shown with a question mark would in fact be the I/O server for the file.) In the current implementation, all streams have shadow streams on the I/O server, but Welch notes that shadow streams use unnecessary memory and should be created only when a stream first becomes shared.

Figure 5.4 shows a shared stream corresponding to the stream `<H,99>` in Figure 5.3. It depicts the stream after process B has migrated from host H to host K. Neither host H nor host K stores the access position for stream `<H,99>`. Instead, they store information about the stream descriptor, which is managed by the I/O server S. The server stores the access position for the stream as well as information about the hosts that have references to it.

Another possible approach to shared access positions is the one used in LOCUS [PW85]. If process migration causes a file access position to be shared between machines, LOCUS lets the sharing machines take turns managing the access position. The operating system on each host agrees to acquire the "access position token" for the file before performing I/O on a file with a shared access position. While a machine has the access position token it caches the access position and no other machine may access the file. The token rotates

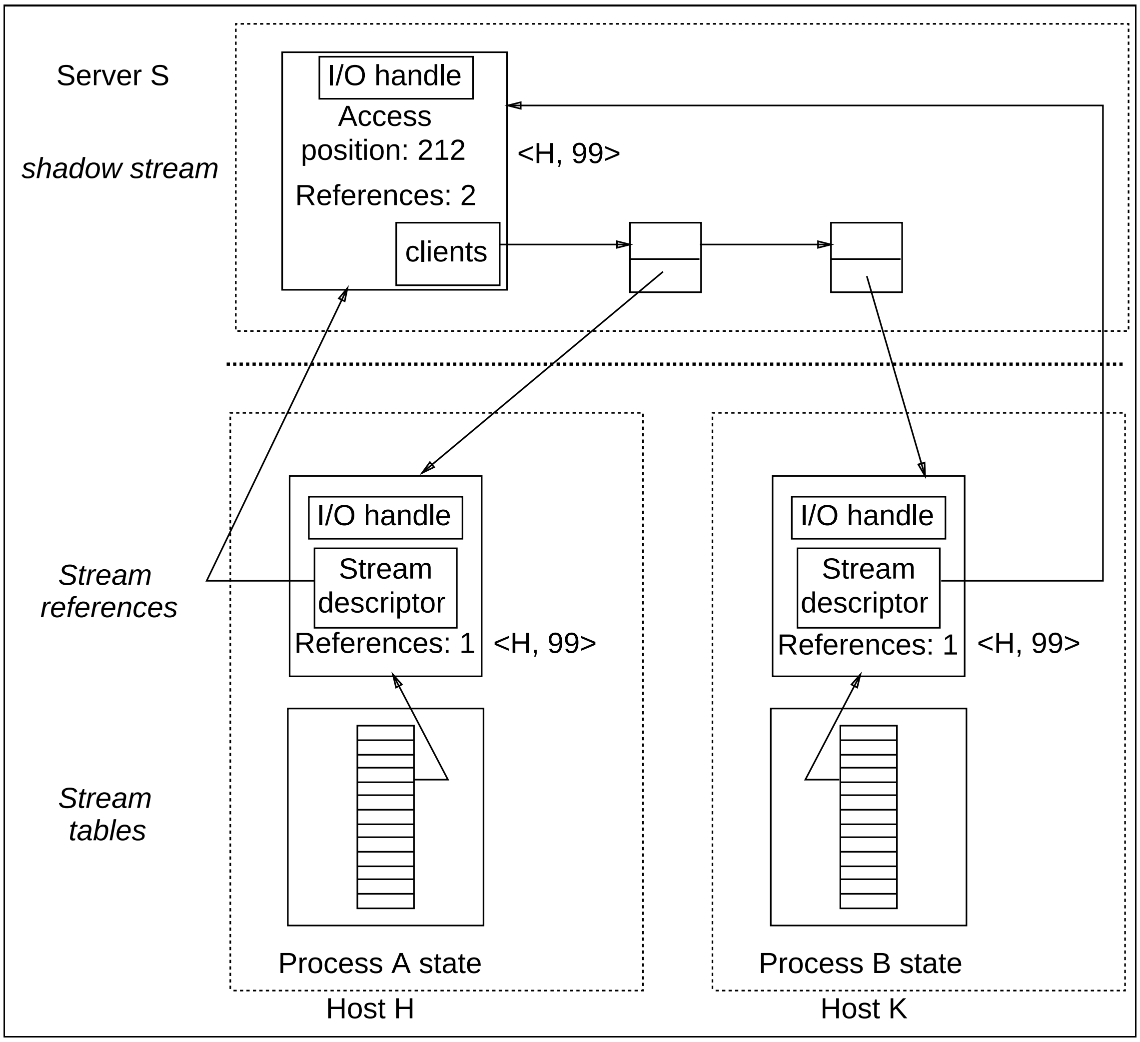

Figure 5.4: *Shadow streams.* This figure elaborates on Figures 5.1 and 5.3. Shared streams in Sprite are supported by forwarding operations on the stream to the I/O server of the file referenced by the stream. The I/O server's copy of the stream, known as a shadow stream, contains the access position for the stream as well as information about each of the hosts using the stream.

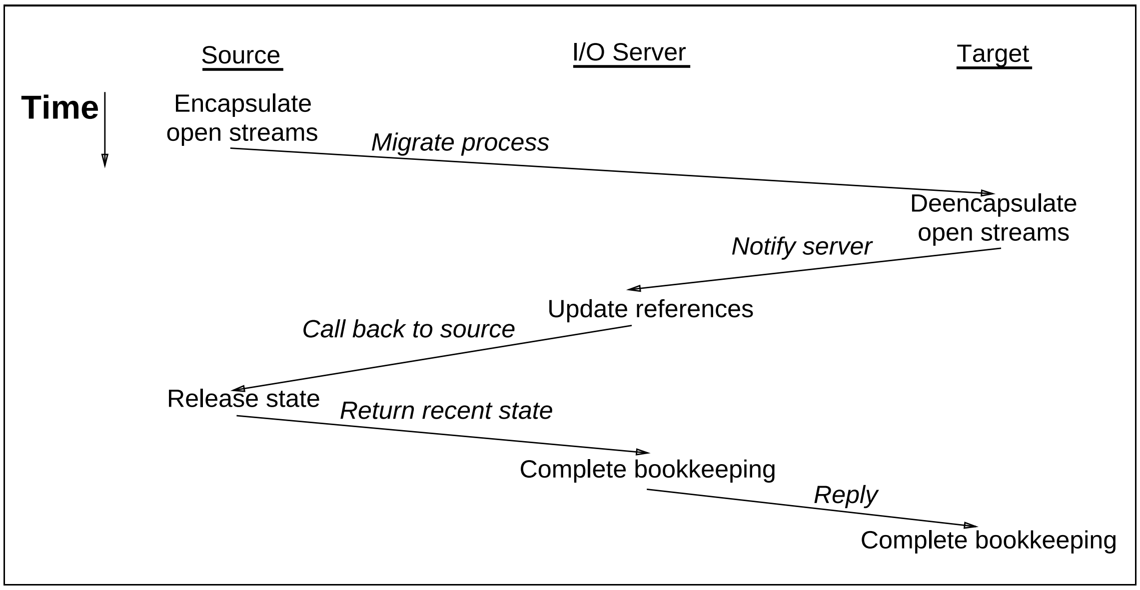

Figure 5.5: *Transferring open files.* The source encapsulates information about all the streams of a process, and it transfers file state to the target when migration is initiated. For each stream, the target notifies the I/O server that the stream has been moved. During this call the server communicates again with the source to release its state associated with the file and to obtain the most recent state associated with the stream. The I/O server then completes its own bookkeeping and responds to the target's request.

among machines as needed to give each machine access to the file in turn. This approach is similar to the approach LOCUS uses for managing a shared file, where clients take turns caching the file and pass read and write tokens around to ensure cache consistency. In his thesis, Welch compares the schemes Sprite and LOCUS use to manage shared access positions; briefly, the LOCUS method requires less network communication if one host performs many operations on a file before another host performs any, while Sprite's method is more efficient if operations from different hosts are continually intermixed. Since we did not believe that sharing would occur often enough to impact performance, regardless of the approach taken, we chose the approach that corresponded to the Sprite cache consistency algorithm: namely, storing shared state on the file server rather than passing a token around.

Atomic Transfer

The first implementation of open-file migration transferred files using existing mechanisms for *open* and *close*, but new migration-specific mechanisms were necessary to avoid needlessly making files uncacheable and to support shadow streams. Figure 5.5 shows the mechanism used by Sprite for migrating open files. The source encapsulates stream state, including the file's access position, into a buffer and transfers the encapsulated state to the target. In the most general case[1], the target machine performs a remote procedure call to the I/O server and requests that the I/O server update its internal tables to reflect that the file is now in use on the target instead of the source. The server in turn calls the source machine to have it release its reference to the stream; the source responds with information about whether any other processes on the source are using the stream. Using that information, the server can determine whether it is appropriate to cache the file on the target, and whether the stream access position should be managed by the I/O server itself or by the target machine.

This two-level remote procedure call synchronizes the three machines (source, target, and server) and provides a convenient point for updating state about the open file. It also permits asynchronous operations on the stream or the file to occur during migration: for example, another process on the source might duplicate or close the stream, or modify the stream's access position. Originally, the access position was encapsulated along with the rest of the information about the stream, but changes to the access position could then be lost once the stream was deencapsulated with its old access position. Now, changes to the access position during migration are handled by retrieving the access position from the source at the time the transfer completes and the source releases its reference to the stream descriptor. The I/O server coordinates other changes to the state of the stream, such as the reference counts it uses for crash-recovery, by locking the stream while it calls back to the source. This makes the transfer of the stream reference atomic with respect to other file system operations. However, locking the stream (and underlying I/O handles) while open files are transferred also complicates synchronization and makes the system vulnerable to deadlock unless considered carefully.

5.4 File Caching

There are two levels of caching in Sprite, one on file servers and one on client workstations. Servers cache file data in their memories to avoid disk accesses, while clients cache data to avoid using the network. While server data caching is important for performance, since it avoids disk traffic, client caching is even more important: caching data on clients not only reduces network traffic, but it reduces server processor utilization as well. As a result, servers can support more clients effectively [Nel88]. In exchange for higher performance and lower utilization, however, the system must face the complexity of keeping the caches consistent.

As I described in Section 5.2.2, the I/O server is the central point for cache consistency operations in Sprite. Nelson described two forms of write-sharing: *sequential* sharing and *concurrent* sharing. If one host writes a file and then closes it, modified blocks for the

[1] It is possible for the I/O server to be the source or target of the migration, in which case some RPC's become local procedure calls.

file are not immediately written back to the file's I/O server. Instead, they remain in the host's cache in the hope that the blocks will quickly be deleted or overwritten. If another host accesses the file while modified blocks are still cached by the first host, the I/O server detects *sequential write-sharing*. It calls back to the first host to write modified blocks back to the I/O server. If two hosts have a file open simultaneously, and at least one host is writing the file, then the I/O server detects *concurrent write-sharing*. In that case, when the sharing begins, any host that is caching the file must flush the file from its cache, again writing any dirty file blocks to the I/O server, and stop caching it. Subsequent reading and writing is performed via RPC's to the I/O server.

Considerations of file caching account for much of the complexity and overhead associated with process migration. First, when a process migrates, its writable files must be flushed from the cache of the source machine. Second, if processes on different hosts write a collection of files that are then read by a process on a single host, those files must be written back to their servers. The following subsections discuss these implications.

5.4.1 Cache Flushing During Migration

Cache consistency has a significant effect on the performance of processes both during and after migration. A variety of factors impact performance; some factors are due to the basic cache consistency algorithm used in Sprite, and some are due to the particular implementation:

- Processes cannot migrate until all modified blocks for their open files have been flushed to I/O servers.
- All transfers within the file system are performed sequentially in 4-Kbyte units, so the effective bandwidth when flushing modified data is substantially less than the maximum RPC bandwidth.
- Blocks are often transferred twice, once from the source to the I/O server and once from the I/O server to the target if they are later referenced.

If Sprite could avoid flushing file data as a side-effect of migration, the performance of process migration could be improved. This change could be accomplished by modifying the basic cache-consistency algorithm to perform consistency on a block-by-block basis; however, such a change might increase the complexity of the file system by a degree that is disproportionate to the improvement obtained. There are a number of simpler improvements that could also reduce the cost of migration. These improvements, described below, are not likely to be implemented unless the impact of file transfer on migration performance increases substantially.

One possible change would be to flush data asynchronously. Rather than making an *open* call or a stream migration wait until data has been flushed to a server, the server could block other operations such as reading and writing. In the meantime, data could be transferred in the background. Migrations would be faster, but after migration a process might have to wait for one or more files to complete being flushed before it could continue.

A second change would be to transfer files more efficiently. For example, the file system transfers data in 4-Kbyte units, which is limited by the maximum RPC bandwidth of 750 Kbytes/sec between two SPARCstation 1 workstations.[2] The maximum bandwidth using 16-Kbyte transfers is 900 Kbytes/sec. Thus, the speed of file transfer could be improved by increasing the unit of transfer. The bandwidth available to flush a single file could also be increased by modifying the file system to use multiple RPC channels simultaneously, as the virtual memory system does. Either of these changes would require significant modifications to the file system, though the resulting changes would benefit all file-system network transfers, not just those related to migration.

A third change would be to transfer modified data directly from the source of a migration to the target, when a file is cacheable by the target. For those files that are read shortly after being written, this change would reduce the number of network transfers required. Transferring file blocks directly would be analogous to changing virtual memory transfer to send modified pages directly to the target, and the same complexity considerations that were discussed in Section 4.2.1 apply in this case.

5.4.2 Cache Flushing Due to Load Sharing

Cache flushing affects not only process transfer, but load sharing as well. Consider a parallel compilation, in which a large number of object files are created simultaneously on separate hosts, and then linked together on a single host. When the linker opens the object files, the I/O server for the files detects sequential write-sharing and causes the files to be flushed to the server. Each file is opened, and flushed, sequentially. The host performing the link obtains the files from the server as individual blocks are referenced.

With the above scenario, it is possible for the cost of file transfers to offset some of the benefit obtained from parallel execution. For example, I measured one benchmark that compiled 139 files in parallel and linked them together. The total size of the object files was 1.2 Mbytes. The time to link the files together when they were all in a single cache was 8 seconds, while the link took 35 seconds when files were spread across 12 hosts. Since Sprite writes modified blocks to the file server every 30 seconds, and the time to compile all the files was approximately 120 seconds, many of the files were undoubtedly flushed prior to the link step. Nevertheless, the time to flush the remaining file blocks and read them into the linking machine's cache was much longer than the time needed to perform the actual link.

The problem with sequential write-sharing due to load sharing is not only that data must be transferred first to the I/O server and then to the client, but also that the transfers

[2] The file system bandwidth between a SPARCstation 1 client and a Sun-4/280 server in practice is 480 Kbytes/sec, due to the slower processor on the Sun-4/280 and to processing overhead within the file system.

start only when each file is opened, so they are performed sequentially. Several techniques are possible for reducing the impact of cache consistency on load sharing; some of these were mentioned above in Section 5.4.1 in reference to improving the performance of migrating a process with cached modified data. Possible solutions include:

- Reduce the delays resulting from opening numerous files that must be flushed. For example, change *open* not to block while a file is flushed, so that applications can open all the files they access without being delayed by file transfers. Alternatively, flush files back to I/O servers more quickly, so less modified data must be flushed at the time files are accessed on new hosts. However, the latter approach would reduce the overall caching performance of the system by causing some data to be written back unnecessarily.

- Make applications more intelligent.

  - Have an application "pre-fetch" files. For example, a program such as the linker could fork one process per file and have its children open the files in parallel. However, children cannot pass file descriptors to their parents, so the parent would have to open the files again, sequentially, once all the *open* kernel calls completed.
  - Have the application "pre-flush" files, instead. Since the delay in opening files results from data being flushed at the time the *open* call is performed, an intelligent application could force output files to their server before they were accessed on another host. For example, *pmake* could open the files it had caused to be created (forcing them to be written back to their I/O servers, assuming they were written by a host other than *pmake*'s) when it had no other work to perform, or applications such as the compiler could flush files to their server before closing them.

Most of the changes listed above would substantially increase the complexity of either the file system or application programs, and none has been implemented to date. The only change that could be implemented trivially is to vary the delay used to determine when to flush a file back to its I/O server. Changing this policy could have a negative effect on system performance in general, despite any improvement in the case of load sharing. Modifying application programs to account for short-falls in the file system is unappealing, though isolating changes in a small number of programs that take advantage of load sharing, such as *pmake*, might make such modifications tolerable. Ultimately, the overhead of cache flushing impacts the speedup obtained from parallel compilation less than other factors such as contention for file servers (as described in Chapter 7). The changes to reduce the overhead of cache flushing should be deferred until cache flushing accounts for a more significant limitation on overall speedup.

5.5 Summary

Supporting time-sharing file-system semantics in a system with many hosts is difficult, and supporting those semantics in the presence of process migration is even harder. Transferring open files between hosts has required careful consideration about cacheability, reference counts, and asynchronous operations on files or streams during migration. In Sprite, the I/O server for a file is ultimately responsible for performing the transfer of stream references in an atomic fashion and invoking any necessary cache consistency operations.

Though file caching provides high performance for the system as a whole, Sprite's cache consistency policy detracts from the speed of process migration. When a process migrates, the migration cannot complete until all modified data blocks belonging to the process's open files have been written to their I/O servers. This delay is exacerbated by the relatively slow rate at which data blocks are written.

Cache consistency actions also affect the performance improvement available from distributing load across multiple machines. If a process on one host accesses data generated on each of several other hosts, the process will be delayed while data blocks are flushed, one file at a time. Without changing Sprite's cache consistency algorithm substantially there is not much the system can do to make existing applications access data faster under those circumstances, but applications can reduce the impact of cache flushing by forcing files to be flushed before they actually need to access them.

Chapter 6

Host Selection

6.1 Introduction

Until now I have focussed on the low-level mechanisms for transferring processes and supporting remote execution. I have described how, given a process and a host to which it should move, Sprite can transfer the process and permit it to resume execution on its new host. However, there is more to load sharing than remote execution. The selection of a target host for migration is also an important mechanism, one that affects the ease and efficiency of migration. There are many possible criteria for deciding when hosts are available, and numerous methods for locating and assigning available hosts. This chapter discusses the goals of host selection and presents several alternative implementations of host selection.

The remainder of this chapter is organized as follows. The next section discusses criteria for host availability. Section 6.3 discusses the goals of a host selection facility, presents several different methods for selecting hosts, and indicates the relative advantages and disadvantages of each approach. Section 6.4 concludes the chapter.

6.2 Host Selection Criteria

In a system with load sharing, a host with more than one runnable process can profitably offload processes onto a host whose processor is underutilized.[1] However, it may be insufficient merely to select any host that has an idle processor at one instant in time. Other considerations include: processor load over a longer interval, host autonomy, hardware configurations, and caching effects.

The most important issue is the determination of when a host is sufficiently underutilized to warrant moving processes onto it. The simplest metric for deciding whether a host has

[1] The same argument holds true for multiprocessors, but this research has been performed only in a uniprocessor environment and is described in those terms.

excess capacity is to check whether it has any runnable processes. However, the number of runnable processes can fluctuate greatly from moment to moment, as I/O-bound processes become momentarily runnable or CPU-bound processes perform a small amount of I/O. The system should therefore consider load over a long enough period that momentary fluctuations will not cause a loaded host to be selected or an unloaded host to be bypassed. At the same time, it must also prevent long-term load averages from blinding the host selection facility to changes in the state of a host. A host that has recently changed from loaded to underutilized may have a high load over the past several minutes, while a host that has just started running a CPU-bound process may appear to be underutilized on average. No single metric in isolation is appropriate.

The trade-offs between using short-term and long-term average loads complicate host selection: the system must balance the desire to use hosts effectively with the need to avoid migrating onto a loaded host, and it must minimize the overhead of making host selection decisions. For example, one possible approach would be to use a small window for averaging utilization (such as a few seconds), but the overhead of keeping the data about each host's utilization current would be substantial. Longer intervals would reduce overhead but make the system less responsive to sudden changes in load. Averages over the past 30 seconds or a minute may be a reasonable compromise between overhead and accuracy, depending on the size of the system and the criteria used for host selection.

If the system can predict future changes in load, it can reduce the inaccuracy of using a longer time-frame. Since the load sharing facility is itself responsible for many of the sudden bursts of processing activity within the system, it can anticipate changes in load by increasing the count of runnable processes by the number of processes expected to migrate onto a host in the near future. MOSIX [BSW89] uses this approach to prevent many hosts from "flooding" an idle host by migrating many processes onto it before its measured load is high enough to prevent further migrations.

The second issue for host selection, host autonomy, arises when the system contains personal workstations. If each host is intended to provide a minimum level of performance to its owner regardless of activity elsewhere in the system, then work should be offloaded only onto hosts that are not actively being used. The minimal criteria for deciding that a host can accept foreign processes would therefore include not only excess processing capacity but also a requirement that a host be idle (*i.e.*, have no keyboard activity) for a predetermined period of time. Furthermore, the longer a host is idle, the more desirable a candidate it is. Mutka and Livny found that hosts that have been idle for a substantial period of time will most likely remain idle for a long time, while hosts that have been idle for a short time are likely to become active again quickly [ML87]. My own measurements, reported below in Section 8.5, support their results.

Other criteria, such as hardware configurations and file caching, can permit a host selection facility to make a more intelligent selection from among multiple available hosts. Hardware configurations can be extremely important if the resources available within a class of hosts vary. Memory sizes are particularly likely to vary from machine to machine.

Since additional memory can make the difference between a large process thrashing virtual memory and running effectively, and can also improve caching performance, hosts with more memory should be selected over hosts with less memory. Other possible hardware considerations include coprocessors or other devices. If some hosts offer performance advantages, using those hosts when they are available would be preferable to using other hosts.

File caching can improve performance if hosts are reused for repeated instances of a single task. If an application reads a fixed set of input files each time it is run, then repeatedly executing the application on the same host will gain the benefit of cached input data (frequently referenced files may remain in the cache indefinitely). Therefore, when selecting a host on which to execute a command, one might consider whether the command recently executed on a host that is currently available. At the same time, independent tasks should use different hosts, so that unrelated processes do not cause the files used by one another to be discarded from the cache.

Table 6.1 summarizes the above criteria.

Criterion	Purpose
Processor load	Avoid contention
Idle time	Respect workstation ownership
Hardware configuration	Obtain higher performance with more memory or special hardware
Past host allocations	Use warm file caches

Table 6.1: *Criteria for host selection.*

6.2.1 The Sprite Policy

Based on the above discussion, one can conclude that there are two components of host selection: identifying suitable candidates and selecting from among those candidates. Processor load and idle time are useful for determining when a host is usable for load sharing. Additional factors, such as differences in hardware and past host assignments, can help the system select hosts with particular performance advantages. This section describes a simple policy Sprite uses to decide when hosts are available and to choose among multiple available hosts.

Host Availability

For a Sprite host to be considered as a migration target, it must meet two absolute minimal requirements: first, it must have less than one runnable process, averaged over the past minute; and second, it must have had no keyboard input in the past 30 seconds. The first requirement ensures that long-running CPU-bound processes already on the host do not compete with new foreign processes for processor cycles. The second requirement prevents foreign processes from degrading the interactive response of persons who are actively using their hosts. The value of the threshold for declaring a host as idle depends on the likelihood of eviction and the overhead of evicting processes; this value in Sprite was decreased from five minutes to 30 seconds after the user community collectively agreed that evictions were not invasive.

These requirements reduce the likelihood of foreign processes competing with local processes for processing, but they do not prevent foreign processes from contending with each other. Sprite prevents contention by allowing only one application of a given class (*e.g.*, compilations or long-running simulations) to migrate onto a host at a time. An application such as *pmake* can obtain several idle hosts for an indefinite period of time, then send a new task to a host each time the previous task using the host completes. (The task can create as many processes as it chooses to, but normally only a single process per host will be ready to execute at any given time.) Allowing an application to reuse a host eliminates the need to select and return hosts for each individual sub-task, which correspondingly reduces the load on the host selection facility. In this sense, Sprite's approach to load sharing is similar to Amoeba's "processor pool," which permits applications to reserve hosts for an extended period of time [MvRT+90].

This technique for avoiding contention is also similar to MOSIX's method of increasing the reported processor load with the expectation that the actual load will immediately increase. However, the adjustment performed by MOSIX is temporary, so if a host has a low load some time after receiving a foreign process, it may accept additional processes. The original foreign process may have to contend with an influx of new processes if it later performs much additional processing, but in that case MOSIX will migrate processes again to reduce the load. Sprite, on the other hand, does not perform automatic migration to correct imbalances in processor load. We chose to err on the side of being too conservative, possibly missing opportunities for multiple applications to share a single processor, rather than complicate the system by handling overloaded hosts.

Selecting an Idle Host

Once a system has decided that a set of hosts have excess capacity, it must choose from among those hosts. If only one host were found to be available, host selection would be trivial. Normally, though, several hosts meet the minimal criteria of processing capacity and idle time described above. The system can choose randomly from among several idle hosts, knowing that any of them would be capable of providing a significant performance improvement, or it can consider differences between hosts in an attempt to gain additional benefits. As an extreme example, the system could use a complicated function of various factors to assign each host a rating of its potential value, and then select hosts in order. A simpler method would be to rank hosts based on one criterion, such as memory size, and then rank hosts within each class on another criterion, such as idle time. Sprite uses a method along the lines of the latter.

In Sprite, we have made some assumptions about our execution environment. First, we assume that any host with a "low" load will execute foreign processes as well as any other host with a "low" load. Second, we assume that the differences between hardware configurations are not significant enough to warrant preferential treatment for some hosts. In our current configuration, all DECstation 3100 workstations are identical, and SPARCstation 1 workstations have between 16 Mbytes and 28 Mbytes of memory. To date, 16 Mbytes of memory have been ample for all the tasks that use migration. (As more applications take advantage of migration, however, we may need to revise this assumption and take memory sizes into account. Similarly, we may eventually need to consider other hardware characteristics, such as floating-point coprocessors.)

With these assumptions in mind, the primary characteristics that distinguish hosts are, first, the likelihood that they will be reclaimed, and second, the contents of their file caches. Minimizing the likelihood of eviction helps both foreign processes and users who return to their hosts, while warm caches can provide performance improvements to some foreign processes. Sprite addresses both of these goals by assigning hosts in order of idle time, with the longest-idle hosts being assigned first. In Chapter 8, I report empirical data to show how this policy significantly reduces the rate of evictions relative to random selection. Chapter 8 also reports empirical data on caching effects. In our environment, assigning hosts in order of idle times tends to assign the same set of hosts to the same processes, permitting them to benefit from warm caches with no additional implementation complexity.

Fair Allocation of Idle Hosts

One problem with reserving an idle host for a single application is that an application might use every host for a prolonged period of time, preventing other applications from obtaining idle hosts. Sprite provides fair allocation of idle hosts in two ways. First, if an application requests a host and none are available, the host selection facility checks whether another application is using significantly more hosts than the one performing the request. If so, a host is reclaimed from the application using more hosts, causing its processes to be evicted to their home machine, and the application that just requested a host is assigned the newly available host. Second, an application specifies whether it expects to use the host for a prolonged period of time. In practice, the system uses two priorities, distinguishing between long-running processes such as simulations and shorter, higher priority processes such as compilations. Hosts that are being used for long-running tasks may still be used over short periods for more important tasks, with the operating system responsible for executing short-lived processes at higher priority than processes that have been executing for a long time. Empirically, these methods permit the Sprite host selection facility to satisfy over 80% of all requests, despite the policy of exclusive access (see Chapter 8 for details).

6.3 Host Selection Mechanism

When a process wishes to offload work, it needs a mechanism for locating idle hosts and, depending on policies imposed by the environment, possibly reserving those hosts so that processes from other hosts do not use them simultaneously. There are a number of possible criteria for the design of a host selection facility. For example, Theimer and Lantz listed three principal goals: performance, scalability, and fault tolerance [TL88]. The *performance* of host selection should be sufficient to impose a minimal impact on processes not using the facility and low latency on scheduling processes that do use the facility. By their definition of *scalability*, a host selection facility should be able to support hundreds of hosts. Finally, to be *fault tolerant*, a host selection facility should be disabled by host failures for at most a few seconds.

I would modify the criteria proposed by Theimer and Lantz to include two additional goals: fairness and simplicity. By *fairness*, I mean that one user should not be permitted to monopolize available hosts to the detriment of other users, as discussed in Section 6.2.1. I will demonstrate that by comparison to the other goals listed above, this condition may place considerable constraints upon the methods used to select hosts. *Simplicity* is of course an important goal of any system, within the constraints imposed by other goals.

Architectures for host selection fall into several categories. They are distinguished by the following attributes:

- **centralization:** Are the data about host availability stored in a single location or on each host separately? Are host selection decisions made by a single entity or by each host separately? In general, centralizing a host selection facility on a single host makes it easier to allocate hosts accurately and fairly, since the agent that selects a host has complete information about the system. However, centralization also makes the facility more susceptible to failures. If the host on which host availability information is stored should crash, then the data will become unavailable. Also, a centralized host may prove to be a performance bottleneck on a large system, since it must manage requests and update messages for hundreds of hosts.

- **accessing host data:** How do processes select hosts? Information about host availability can be kept in one or more shared files, or it can be stored in the memory of one or more server processes. Using the file system is simpler than using processes, because the file system can provide synchronization and permanence, but it also has several deficiencies as I shall describe.

- **state:** Does the host selection facility retain state about past assignments? Maintaining a history of previous assignments may be useful to take advantage of warm file caches, and it is also necessary for reclaiming hosts due to fairness considerations.

Host selection facilities have been discussed in detail in the literature. Most or all of them use server-based approaches, because they were designed for systems without a shared file system. For example, Zhou compared several algorithms for load balancing, including a centralized server, a broadcast-based distributed system, and a hybrid system in which each host sends its load to a central coordinator, which periodically broadcasts the load of every host in a single message [Zho87]. Theimer and Lantz compared implementations of a centralized server and a distributed request-response protocol using multicast (which permits one message to be sent to many recipients without an undue burden on hosts not receiving the message) [TL88]. They found that a centralized server can service thousands of clients efficiently if status update messages are limited to idle hosts, whereas a distributed implementation using broadcast or multicast results in a high demand for network bandwidth that would limit a system to a few hundred hosts. Also, in their implementations of decentralized and centralized scheduling, decentralized scheduling took two to four times longer than centralized scheduling, though it was still "acceptably short". Their overall conclusion emphasized simplicity over scalability and performance, however: with the availability of efficient multicast, they believed a decentralized design is simpler than a centralized one.

However, much of the past work on host selection is only partially applicable to the Sprite environment. In our environment, host selection operations are less frequent than they would be in a system that dynamically migrates processes continually to balance load; no multicast is available; and we have a shared network-wide file system. Sprite is of course not alone in these respects. For example, Butler used a shared file to store the state about each host; later, for improved performance, Nichols changed Butler to use a centralized process [Nic90]. Condor also uses a centralized process to control access to idle hosts [LLM88]. This thesis supports Nichols's and Litzkow's conclusions with respect to centralization, as well as examining additional issues such as fair allocation of hosts.

In this section I compare four host selection architectures using the criteria listed above. The first three are implemented in Sprite, and the descriptions of these architectures focus on the Sprite implementations. First, Section 6.3.1 describes a way to use a shared file to store information about each host. Processes make their own determination of what hosts are available by reading the file to obtain the status of each host. Second, Section 6.3.2 demonstrates how a centralized server process improves upon the shared file approach in most respects except simplicity. It is particularly appropriate for addressing any need for globally fair host allocation. Third, Section 6.3.3 considers techniques that distribute host information across multiple hosts to avoid processor contention and increase reliability. Finally, Section 6.3.4 describes a stateless mechanism that uses multicast to send requests for hosts, and then selects from those hosts that respond first [The86, TL88]. Table 6.2 summarizes the characteristics of each architecture.

6.3.1 Shared File

If host selection is performed using a shared file, then the information about host availability is centralized but selection decisions are distributed. With this technique, each host runs a daemon process that keeps track of the characteristics of its own host and stores that information in the shared file. This organization is depicted in Figure 6.1. The information

Name	Centralized?	Message or File?	Retains State?
Shared file	yes	file	yes
Central server	yes	messages	yes
Distributed servers	no	messages	some
Multicast request	no	messages	no

Table 6.2: *Architectures for host selection.*

must be updated any time important data in it changes (for example, if a host becomes idle after being active or vice-versa). By associating a time-stamp with the information for each host, and updating the information periodically even if it has not changed, processes reading the data can detect out-of-date data.

Sprite initially used the shared file approach to select hosts. With this implementation, each record in the shared file is fixed in length and consists of ASCII characters separated by spaces. A record contains the following fields:

- an identifier for the host associated with the record,
- the host's average load, over periods of 5–15 minutes,
- the time the host last rebooted,
- the time the record was last updated,
- the time since the host last had interactive input,
- an indication of whether the host will accept foreign processes,
- the number of applications assigned to the host (either 0 or 1 in the file-based implementation, which did not support multiple priorities of requests), and
- the version number of the host's migration facility.

With this information, each requesting process can perform host selection decisions individually. A process reads the file and finds all hosts with that are accepting foreign processes and are not already assigned. (It excludes any hosts with a different migration version or whose timestamp is old enough to suggest that the host is no longer running.) The process selects an available host, then rewrites the file to update the number of applications using the host.

The shared file approach has the advantage of being extremely simple. It uses existing facilities in the file system to provide synchronization: a process will lock the file while it accesses it in order to serialize operations on it. Existing facilities also provide recovery in the face of host failures. If the host that stores the file fails, the shared file becomes

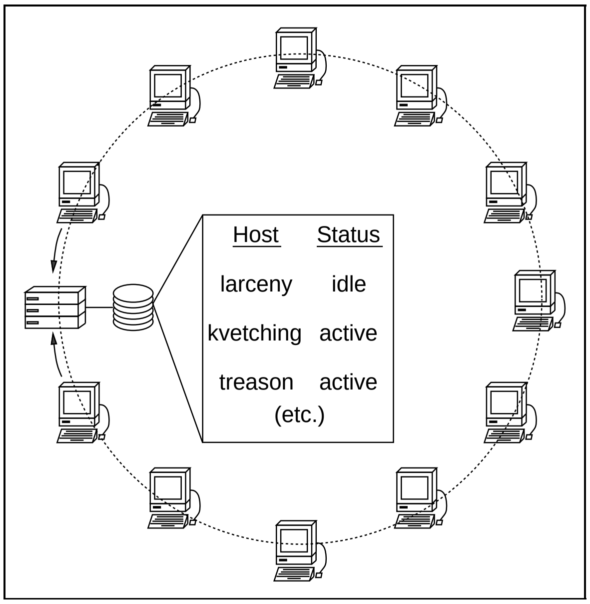

Figure 6.1: *Host selection using a shared file.* Daemons on each host use normal file system operations to update their state periodically in a file.

unavailable, but much of the Sprite file system also becomes unavailable so the temporary loss of the host information database does not additionally affect operation. When the host returns to normal operation, the operating system on each host will recover state about the shared file and permit processes to continue to access it as before. If another machine fails while accessing the file (*i.e.*, while the file is locked), standard file system procedures ensure that the host's failure does not affect future access to the file.

Note that if the file system supports replicated files, then the failure of a single host will not make the host availability database unavailable. The host selection facility could be made much more reliable by storing data redundantly, but the cost of updating the data during normal operation would be commensurately higher. On the other hand, there would be no additional implementation complexity within the host selection facility to support replication, since the file system would provide the underlying mechanism. With respect to reliability, a shared file in a replicated file system might then compare favorably to a replicated, distributed server-based architecture (described below in Section 6.3.3).

Because of its simplicity, the shared file approach also has a number of disadvantages. Its greatest problem is the number of network communications required for operations such as host selection. Each of the following operations within the file system requires a remote-procedure-call:

1. The process *opens* the file containing host information.

2. The process *locks* the file to prevent other processes from accessing it simultaneously.

3. The process *reads* the file to obtain the host information. This operation can potentially require multiple network transmissions since the file may be large (in Sprite, each host entry in the file is 169 bytes, so a maximum of 8 records can fit in one 1500-byte Ethernet packet).
4. The process selects an idle host and *writes* the file to flag that host as in-use.
5. The process *unlocks* the file.
6. The process *closes* the file.
7. After finishing with the host, the process *opens* the file,
8. *locks* it,
9. *writes* it to indicate the host is available,
10. *unlocks* it, and finally
11. *closes* it.

The total number of file system operations necessary to select and release a single host is thus 11, including 10 "small" RPC's (transferring 100-200 bytes at most) and one or more "large" RPC's (transferring several Kbytes). If a process selects multiple hosts, then it can open and close the file only once during that time, and it can potentially obtain multiple hosts with a single read operation. Each independent request within the same process still must start with locking the file and conclude with unlocking it. Similarly, confirming that a host is idle before migrating a new process onto it requires that a process lock the file, read a single record, and unlock the file.

The shared file approach has other disadvantages in addition to performance. It is used by unprivileged processes, so it must be made readable and writable by all users. Also, even if the shared file contained enough state for an application to determine that a host should be reclaimed from another process, the application would have no capability to reclaim the host itself. Finally, the shared file approach requires explicit deallocation of hosts when they are no longer used by an application, so the abnormal termination of a program using idle hosts can leave a host marked as in-use for a prolonged period of time.

In order to address the disadvantages of using a shared file, one can store host availability data in the memory of a process rather than a file. The next subsection describes the implementation of a centralized process-based host selection facility in Sprite.

6.3.2 Central Server

The current host selection facility in Sprite uses a centralized server process to store host availability data in memory. As with the shared file approach, a daemon on each host periodically updates information in a centralized location, but the updates are sent to a

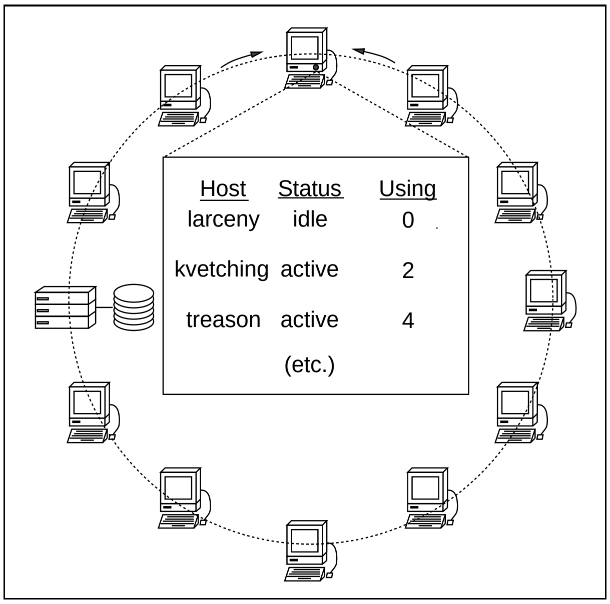

Figure 6.2: *Host selection using a central server.* Daemons on each host send updates to a process on one host. The state of each host is stored in the memory of the process. The process assigns hosts to processes based on the state of each host, including outstanding allocations of hosts to processes on other hosts. In this example, processes on "treason" are using 4 idle hosts, so one of those 4 hosts could be reclaimed if a process on another host requested an idle host and no more hosts were available.

---

process rather than stored directly in the file system. This approach is shown in Figure 6.2. In some respects, using a centralized process shares the simplicity of the shared file approach: all the information is current, and each host need only communicate with one other host. The primary distinction between the two approaches is that the protocol for requesting a host from the central server requires fewer network communications.

1. The process *initiates* communication with the central server. In Sprite, this operation is performed using the file system: the process opens a special file, known as a *pseudo-device* [WO88], and the operating system passes the open operation on to the server. Communication using a reliable stream protocol, such as TCP/IP, is another possibility.

2. The process *requests* one or more hosts. In Sprite, the request takes the form of a file system kernel call, *ioctl*, but other forms of communication (*e.g.*, read/write, UDP/IP, or RPC) would be appropriate.

3. After finishing with the host, the process *returns* the hosts to the pool of idle hosts, and

4. *closes* its communication channel. Note that hosts are returned implicitly if communication is terminated while hosts are still assigned to the process. Only if a process wishes to retain some hosts while releasing others must it explicitly communicate with the server to return hosts.

In Sprite, the total number of file system operations necessary to select and release a single host using the central-server approach is 3: one to open the pseudo-device, one to send the request for a host, and one to release the host or close the pseudo-device. The *open* file system call would typically require two RPC's, one to resolve the name corresponding to the pseudo-device and one to contact the I/O server for the server. None of the RPC's transfer an especially large amount of data: the largest would be the pathname for the pseudo-device. On average, host selection by a single Sprite process using a central server and pseudo-devices takes 22 milliseconds on a SPARCstation 1 workstation.[2] The central server can support a maximum of 67 request/return pairs per second when multiple hosts make requests in parallel, during which time the processor on the host running the server is 99% utilized.

A centralized server has other advantages over a shared file besides performance and scalability. Using the connection-oriented protocol described above, the server can detect when a process that is communicating with it terminates. It can then free the hosts that process was using if the process had not already explicitly released them. Note, however, that the overhead of establishing and terminating the connection is significant; a connectionless protocol would require fewer network operations to request and release an idle host, though detecting a defunct application would be more difficult.

Another advantage of a centralized server is that it can communicate asynchronously with processes that are using idle hosts. The server can inform a process that a host it was using is no longer available, either because of eviction or because the host was reclaimed due to fairness considerations. It can also notify the process when a new host becomes available. In a message-based system, the server would just send a message to the process; in Sprite, a more complicated approach is necessary. Processes communicate with the central server using the *ioctl* kernel call, but performing an *ioctl* when no change in status has occurred would generate needless overhead (in particular, a context switch to execute the central server). A *select* kernel call, on the other hand, can ask the kernel whether a stream is readable, and does not require any interaction with the server. (In the current implementation, *select* does require a kernel-to-kernel RPC to the I/O server for a file or pseudo-device; in principle, even the RPC could be avoided by having the I/O server notify the user's kernel once a stream becomes readable.) Thus, to notify a process of a change in status, the server makes the process's stream readable. The next *select* kernel call by the process will indicate that a change is pending, at which point it performs an *ioctl* kernel

[2]Note that it is not possible to compare the central server and shared-file approaches using the same configuration, since the shared file implementation in Sprite has been decommissioned. The time to select and release an idle host using DECstation 3100 workstations was measured to be 56 milliseconds. [DO91].

call to obtain the data from the server. Normally, no changes are pending and the stream is not readable, so no network operations should be needed.

However, centralizing host selection in one process has two potential disadvantages as well: it makes the server a single point of failure, and it loads down the host on which the server executes. In addition, a centralized process may not scale well if a connection-oriented protocol is used. The remainder of this section discusses these issues.

Reliability

In the case of a shared file approach, the failure of the file server storing the file can disable access to the host availability database. A central server is similarly vulnerable to the failure of the host on which the process runs, and it is also vulnerable to software faults in the server process itself. Theimer and Lantz noted that there are two reasonable alternatives for making host selection tolerate failures [TL88]. The facility can be replicated, so that one failure does not disable the facility, or the facility can be restarted as soon as its failure is detected.

Replication requires additional complexity and run-time overhead, and is discussed below in the context of distributed servers. If the host selection facility need not be constantly available, though, it is simpler and cheaper to recreate it after a failure. Nevertheless, the "reinstantiation" approach has potential problems that must be addressed:

- A process must notice that the server has become unavailable and then create a new one. If many processes notice simultaneously, they may compete for the opportunity to create the replacement. Sprite addresses the problem of contention in three ways. First, the load average daemon on each host monitors the state of the central server. If a daemon detects that the server has failed, it waits a random period of time, and then attempts to create a new instance of the server if the server has not yet been reinstantiated. Second, the file system provides automatic mutual exclusion, so that only one process can manage the file system object (pseudo-device) that other processes use to interact with the server. Third, the server periodically checks whether another process has inadvertently removed and recreated the pseudo-device; if so, the server exits and permits the later instantiation to control host selection instead.

- The state maintained by the old server is lost. The new server will not know about outstanding host assignments, nor will it know about the state of each host. The daemons running on each host must therefore communicate with the new server to restore its state.

- Using a connection-oriented protocol, such as Sprite's, processes with connections to the old server will encounter error conditions when they try to communicate with it. They, too, will need to initiate new sessions with the new server. Since interactions with the server are hidden behind a library interface, connections are reopened automatically when an error occurs. An application that uses the host selection facility need not be aware of the error condition.

Load

The other potential disadvantage of using a centralized server is the load it imposes on its host. Theimer and Lantz considered a system in which the server runs only on idle hosts, and migrates (or is recreated) when the host on which it executes becomes unavailable. In that case, resource utilization is unimportant as long as the processing power of the host is sufficient for the server to respond to all update messages from the per-host daemons and all requests to obtain or return idle hosts. Though restricting the central server to run on an idle host is certainly feasible, it is an unnecessary requirement if the processor and memory requirements of the server are minimal enough not to impact the response of active users on the same host.

As was mentioned above, the server's processor can be completely utilized if it receives a steady stream of requests for hosts. Since applications cache host assignments to reuse the same idle host repeatedly, however, the actual rate of requests is currently quite low (3.25 requests per minute). This point is discussed further in Section 8.5.

Scalability

Using a connection-based protocol, the centralized server must store on-going state about every process that communicates with it. One may wonder how the accumulated state per open connection affects the scalability of the centralized server. In the Sprite implementation, the limits on scalability arise not from the state maintained by the server, but from the state maintained by the operating system for open pseudo-device connections. The size of the state stored by the Sprite central migration server for each open connection is 96 bytes. Since this state is stored in virtual memory, the host could handle thousands of client processes (and per-host load average daemons) with no adverse effects. However, the operating system stores an I/O handle for each outstanding process as well. Each I/O handle is 100–200 bytes and is stored in physical memory. If kernel memory proves to be too valuable a resource to permit long-term pseudo-device connections to the host selection server, then a connectionless protocol will be required.

6.3.3 Distributed Servers

In order to avoid the potential problems of a centralized host selection facility with respect to availability and processing requirements, the facility can be distributed among multiple hosts. There are two approaches toward distributed host selection: "advertising" and "query-based." With advertising, each idle host periodically updates its state on each of the hosts running the host selection facility, much as is done with a single central server.

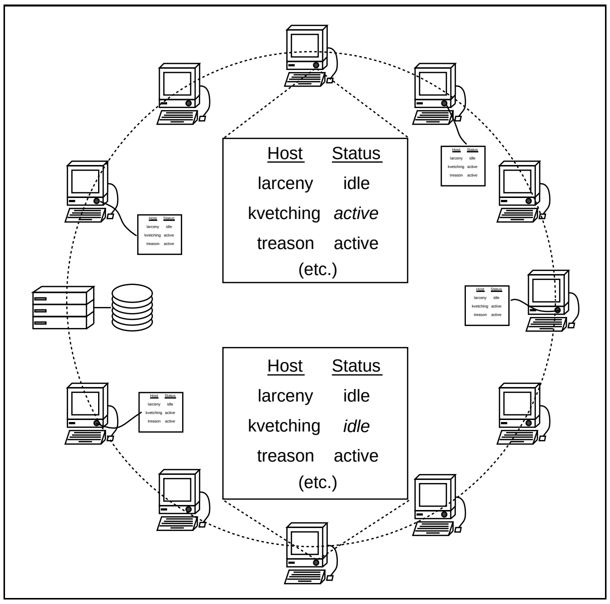

Figure 6.3: *Host selection using multiple servers.* Servers on some or all of the hosts keep track of the state of other hosts. The distributed approach addresses some of problems the centralized-server approach, such as fault tolerance and scalability, but keeping the state of the system consistent across different servers is difficult. For example, one host might think the host "kvetching" is idle while another thinks it is active.

With a query-based approach, a host with processes to offload sends a multicast (or broadcast) message requesting idle hosts, and hosts that are available respond directly to the requesting host. In either approach, there is no central point of failure and no single host that must service large numbers of requests; however, other issues arise, such as the need to keep state consistent on multiple hosts. This section discusses the advertising approach, and Section 6.3.4 considers query-based host selection.

Using the advertising approach, the host selection facility can run on some or all hosts. If a host selection server runs on every host, then processes can obtain idle hosts by contacting the server on their own host. If a host later fails, its host selection server can be restarted when the host reboots. However, software failures must still be detected, so that a server can be restarted even when its host has not rebooted. If not all hosts run a server to keep track of host availability, then the system must provide a mechanism for processes to contact an appropriate server, as well as a provision for replacing failed servers.

Regardless of how many hosts run host selection servers, two problems arise. First, if servers on different hosts execute independently, then they may have inconsistent or outdated views of the state of the system as a whole. (This situation is shown in Figure 6.3.) When a server on one host allocates a host, how do other servers determine that the host has been allocated? If a host is returned to the pool of idle hosts, how quickly do servers learn that the host is available? In the former case, an application may attempt to migrate to a host that has already been claimed; in the latter, the application might miss an opportunity to use an idle host. It may also be difficult to ensure that hosts are allocated fairly, since decentralized control suggests that no one server knows the exact state of the entire system at a given time.

The second potential problem of distributed servers is processing overhead. Consider a system of several hundred hosts, with hosts updating their state an average of once per minute. Assuming service times on the order of a few milliseconds, the processor overhead of servicing update messages would be about 1% per hundred hosts. The overhead of processing requests and keeping information among servers consistent would add to that cost. Thus, the processing overhead of the entire host selection facility might be on the order of 5% of processing capacity. Theimer and Lantz note that this cost greatly detracts from the desirability of having all hosts perform their own scheduling decisions, since even busy hosts must pay the cost of processing incoming update messages [TL88].

Two variations on the distributed approach address these problems. With probabilistic host selection, each host periodically updates state on a random subset of other hosts. This method, used in MOSIX [BSW89, BS85], reduces overhead but still suffers from distributed state. Another possible approach is a replicated fault-tolerant facility that would run on only a few unloaded hosts, such as the facility currently being implemented in the ISIS system [Bir85, Coo90]. The remainder of this section describes these two alternatives.

Probabilistic Host Selection

Barak and Shiloh described a distributed host selection facility for MOS (later MOSIX) that used a probabilistic algorithm to disseminate host information [BS85]. Each host kept track of the most recent information it had about other hosts. In order to avoid using obsolete information, a host would "age" old data as newer data arrived, thus giving more weight to recent data than to old data. At regular intervals, each host would randomly select another host and send that host its own load as well as the newest information it had received about other hosts.

The original mechanism for MOSIX suffered from a propagation delay that resulted from indirect data. It would take several propagations for a substantial change in a host's load to become apparent to all other hosts. MOSIX currently propagates load information by exchanging data directly with other hosts: a host selects three another hosts, sends its load data to those hosts, and obtains the load of those other hosts in return. Loads are exchanged once per second in order to ensure that hosts have fairly recent information about all other hosts. The value of a host's load is artificially modified to include anticipated fluctuations due to migration [BSW89]. However, substantial changes in the load of a host still take several seconds to propagate to other hosts, and in the meantime hosts with obsolete load information may attempt to migrate onto a loaded host.

Stolcke and von Eicken [SvE89] built a distributed host selection facility for Sprite based on the original MOSIX model. They then compared their distributed probabilistic facility with the shared file approach that was used in Sprite at the time. They found that by using their facility, the service time for requests for idle hosts remained relatively constant, regardless of the overall rate of requests throughout the system. On the other hand, the shared file was a significant bottleneck when idle hosts were requested rapidly on many hosts simultaneously. The centralized server process is a similar bottleneck, as mentioned above.

The probabilistic approach reduces contention for processing time under periods of high load, but it also suffers more from incomplete knowledge under those conditions than during periods of low load. Each host would likely have an outdated view of the state of other hosts, and a high rate of requests would cause those states to fluctuate rapidly. As a result, a host servicing a request might have no knowledge of the availability of a host whose load had just dropped. Furthermore, since no host has complete knowledge of the state of the system, it is difficult for any one host to make decisions that affect overall state.

Replicated, Fault-tolerant Servers

There are two problems with the distributed host selection facilities described above: no single server has complete knowledge of the system, and loaded hosts must process update messages to know the state of other hosts. Limiting the host selection facility to a small number of lightly loaded hosts would address the latter problem but not the former. In addition, if processes do not simply request idle hosts from a server on their own host, they need a mechanism for locating a server. A system such as ISIS [Bir85], which provides support for replicated fault-tolerant applications, can address these issues.

ISIS permits a distributed facility to be logically centralized, yet composed of fault-tolerant components executing on multiple hosts. A logically centralized facility would have current knowledge of the state of all hosts and any outstanding host assignments, so it could make accurate decisions regarding the availability of hosts and the fairness of existing allocations. Since each process would react to each message, this facility would have essentially the same performance characteristics as the physically centralized server described in Section 6.3.2. A prototype host selection facility using ISIS is currently being developed at Cornell University [Coo90].

6.3.4 Multicast Requests

Another way to address the problems of a centralized repository for host information, namely processor contention and reliability, is to have no repository at all. Instead of having hosts announce their state in advance, they respond to requests for idle hosts if they are available. Querying hosts individually is not feasible, because the latency to find an idle host that is not already in use could be substantial. Instead, using multicast, a process can send a single message and only those hosts that wish to receive it can do so. If more than one host responds, the querying process can select from among the responders. Later, to

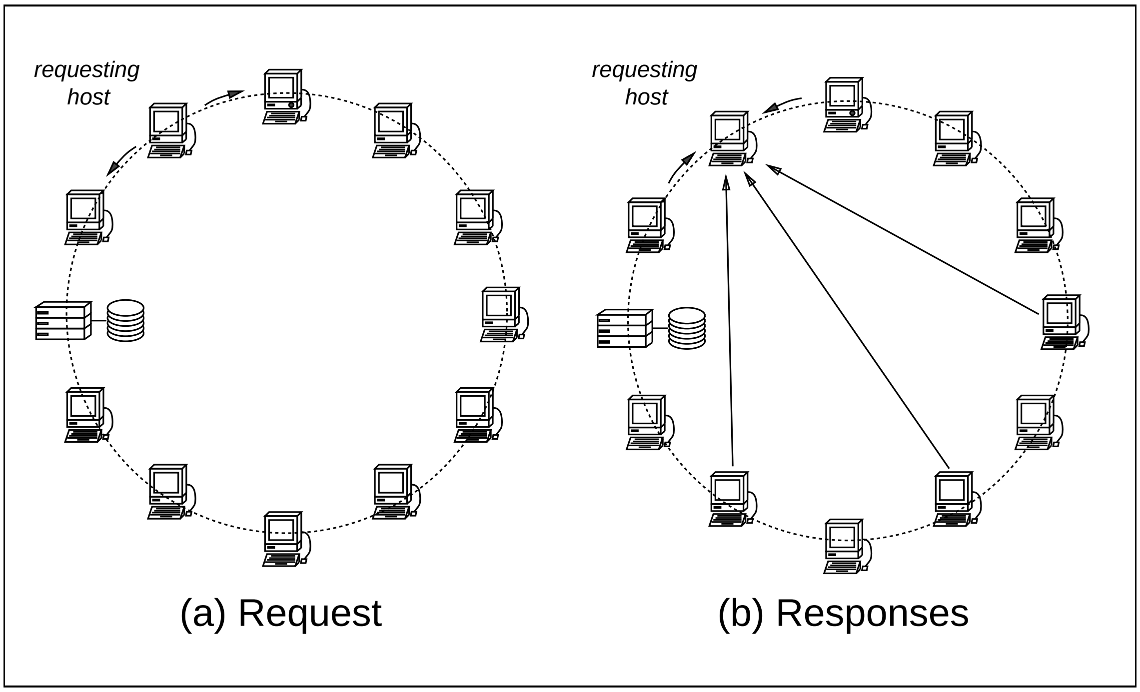

Figure 6.4: *Query-based host selection using multicast.* Figure (a) shows a host sending a request to all other hosts. In (b), those hosts that receive the request and choose to respond send back a message to indicate their availability. The requesting host selects from those hosts that respond.

reuse a host, the process would contact the host directly to obtain its current status. The query-based approach is demonstrated in Figure 6.4.

Host selection using the "querying" approach is completely distributed and is unaffected by the failure of any subset of hosts. The greatest problem with this approach is the number of messages generated by each request when many hosts are available. Since each request can result in as many responses as there are hosts, the network may become temporarily saturated by responses. Theimer and Lantz estimated that a decentralized facility based on this design scales to at most a few hundred hosts, while a centralized architecture using stateless communication could handle thousands of hosts [TL88].

A second problem with the querying approach is that there is no global information about the state of the system or previous host assignments. In particular, Theimer and Lantz noted that the decentralized approach does not consider global fairness, which can become an issue if more "long-running and/or massively parallel applications become more prevalent." As my measurements in Chapter 8 show, long-running applications are becoming common in Sprite, as are applications whose parallelism is constrained only by the availability of idle hosts.

6.4 Summary

The most important aspects of host selection are the policy for choosing among available hosts and the mechanism for assigning hosts to processes upon request. There are many possible criteria for selecting one host over another: for example, idle time, load, and previous host assignments. There is a strong correlation between idle times and the likelihood of eviction; since the cost of eviction is likely to outweigh the benefits of slight decreases in load or of using warm caches, Sprite allocates hosts in decreasing order of idle time.

The goals of a host selection mechanism include performance, reliability, scalability, fairness, and simplicity. Global fairness is an important aspect of load sharing in any environment in which long-running processes are common. Fairness is also one of the most difficult goals to address, and most existing host selection facilities have not considered fairness to be a compelling issue.

With those goals in mind, I considered several approaches to host selection, varying along a number of parameters. The two simplest approaches, a shared-file repository and no repository at all, are inadequate: the former has poor performance, as well as insufficient security, and the latter does not scale and does not ensure fair allocation of hosts. Probabilistic host selection scales well with respect to performance, but it allocates hosts using incomplete information and has no global state. A centralized architecture is relatively simple and is easy to reinstantiate in the case of failure. It is appropriate for addressing the issue of global fairness, and it provides adequate performance unless requests are made extremely frequently. For these reasons, the centralized facility described in Section 6.3.2 is currently used in Sprite. However, if better fault tolerance is needed, a replicated, logically centralized facility can improve the reliability of a centralized server while still allocating hosts fairly.

The choice of a centralized or distributed host selection facility may ultimately be based on the relative importance of global fairness and frequent host selection. Naturally, if Sprite's host selection facility had to respond to more than 67 requests per second in normal operation, a central server would be unable to satisfy demand. Any significant fraction of that demand would be too great to permit the host selection facility to run on a user's workstation, for that matter. In practice, though, we have found that global fairness is more important than avoiding a bottleneck on the server's processor. Chapter 8 reports on the low average rate of requests for idle hosts and the high rate at which hosts have been reclaimed due to fairness considerations. In recent months, as many hosts have been reclaimed to reassign them to another application as to return them to their owners.

Chapter 7

Performance

7.1 Introduction

The performance of process migration depends on many factors, such as the availability of idle hosts, the time to transfer a process, the overhead of supporting transparent remote execution, and the frequency of evictions. This chapter reports the measured costs of migration in Sprite and the performance improvement obtainable by typical applications, and the next chapter reports empirical data on overall usage patterns and average costs in day-to-day use.

Section 7.2 briefly summarizes the implementation of process migration in Sprite and its common usage.

In Section 7.3, I use a collection of "micro-benchmarks" to analyze the constituent costs of migration in Sprite, such as the time to migrate a trivial process, transfer an open file, or select an idle host. These measurements collectively demonstrate that the overhead of process migration is minimal.

In Section 7.4, I examine the effect of process migration on the performance of compilations and simulations, which are the primary applications that use migration in Sprite. The speedup obtained depends in large part on contention for system-wide resources such as file servers and the rate at which new processes are generated. For example, an application that creates many processes that interact with a file server and then exit can saturate both the file server and the host running the application. As a result, it obtains a speedup of only 3–6 even when as many as 12 hosts are used in parallel. An application that creates a small number of processes that execute for a relatively long time and do not compete for a file server can obtain much higher speedup.

Finally, Section 7.5 draws some conclusions based on the measurements presented in this chapter.

7.2 Implementation Summary

Process migration in Sprite is divided into two parts. The *mechanism* of migrating processes between hosts and supporting transparent remote execution is summarized in Section 7.2.1. The *policy* of selecting idle hosts and evicting processes when hosts are reclaimed is summarized in Section 7.2.2.

7.2.1 Mechanism

Processes can migrate between hosts at any time. When a process migrates, much of its state is transferred from the *source* of the migration to the *target* machine. The state transferred includes open files (requiring that modified data blocks in the cache of the source be flushed to their I/O server), signal handlers, identifiers, and other state that depends on the type of migration. If the process migrates at an arbitrary point, its modified virtual memory pages are written to a shared backing store, and the process demand pages its memory image after migration. More frequently, a migrates in conjunction with the *exec* kernel call. With *Exec*-time migration (also known as remote invocation), the process's execution image is completely replaced, and no virtual memory is transferred. However, the arguments to the program invoked are transferred to the target, as are the process's environment variables.

No matter where a process physically executes, it behaves as though it executes on a single host throughout its lifetime. That host, known as its *home machine*, is where it would execute in the absence of migration. To support transparent remote execution, a few location-dependent system calls are forwarded from the kernel of a remote process to the kernel of its home machine. Some calls, such as *fork*, are processed by both the remote host and the home machine. Operations upon the process, such as signals, are forwarded from the home machine to the process's current location.

7.2.2 Policy

Policy in Sprite is managed by cooperation between a central server and application programs (*pmake*, in particular). The central server keeps track of idle hosts and assigns a host to one application at a time. Application programs open a connection to the server, request one or more hosts, migrate processes onto the hosts, and return the hosts to the pool of idle hosts once they are no longer needed.

If a user returns to a host that is running foreign processes, a load-average daemon on the host *evicts* the foreign processes, migrating them back to their home machine. The daemon notifies the central server that its host is no longer available. The central server, in turn, notifies the process that had been assigned the host. That process can permit the evicted processes to run on the home machine, remigrate them to another idle machine, or suspend them.

If one application uses more than its fair share of idle hosts, the central server can reclaim some of the hosts used by the application. In that case, foreign processes on the host are evicted, just as if the user returned.

If a host is reclaimed, or a new host becomes available, the central server notifies any affected processes. The notification is performed in two stages. First, the server makes the processes' streams readable. When a later *select* kernel call indicates that a stream to the server is readable, an application process performs an *ioctl* kernel call to obtain the information from the central server. Application processes are expected to perform the *select* call before reusing an idle host for repeated migrations. *Pmake* also uses the notification mechanism to learn of evictions and remigrate or suspend evicted processes.

7.3 Constituent Costs

The overhead of performing process migration may be broken down into three parts: host selection, state transfer, and forwarding of kernel calls. Host selection occurs prior to migration, state transfer occurs as part of migration, and system calls are forwarded subsequent to migration when a process moves to a remote host and executes location-dependent kernel calls. State transfer is the most costly aspect of migration. It consists of fixed overhead to transfer data such as process identifiers and to allocate and initialize a new process, as well as the additional time necessary to transfer open files; if the migration is not performed at *exec*-time, then virtual memory must be transferred as well. Though the costs of host selection and forwarded kernel calls do not typically affect execution time to a large extent, either can be costly if performed frequently. The remainder of this section discusses each of the component costs in detail.

7.3.1 Host Selection

The cost of host selection in Sprite consists of several components: the time to open a connection to the central server process, the time to select one or more hosts, the time to confirm that hosts are available before reusing them, and the time to return hosts to the pool of idle hosts. In the normal mode of operation, a process will establish a connection once and request a number of hosts before closing the connection (implicitly returning the hosts). If it reuses the same host for multiple remote invocations over time, the application is expected to confirm that the host is still available each time it uses the host. As described in the previous chapter, checking the availability of a host requires only that the application perform a *select* kernel call, which indicates whether the central server has a information for the application about a host's changing state. (Note that the application relies on the central server for update messages, rather than communicating directly with the host being used; this is due to the separation between the migration mechanism, implemented by kernels, and the load sharing policy, implemented by the central migration server.)

On SPARCstation 1 workstations running Sprite, the cost of communication with the central server is dominated by overhead for file system lookups and pseudo-device commu-

Action	Time (milliseconds)
Open connection to server	20
Obtain one host	11
Close connection to server	5
Total	36
Obtain N hosts one at a time	$(25 + 11N)$
Obtain N hosts with one request	$(36 + 0.1(N - 1))$
Confirm that a host is available	2

Table 7.1: *Costs of host selection.* All measurements are the average of 1000 iterations, performed on SPARCstation 1 workstations. The total time to initiate a connection and select a single host corresponds to the worst case of host usage, when the full protocol is required for each migration. The start-up overhead may be amortized across multiple requests or by requesting multiple hosts in a single operation.

nication. Opening a file takes about 6 milliseconds if the I/O server of the file is the same as its name server; for the pseudo-device for the migration server, an additional 5 milliseconds are necessary to contact the host running the server, plus about 5 milliseconds to copy data into the migration server's address space and context-switch the server to make it runnable. The total cost of contacting the central server is 20 milliseconds, of which roughly one-fourth is due to network communication costs and three-fourths are due to processing overhead (roughly 150,000 instructions assuming a 10 MIPS machine).

Once the server has been contacted, it takes 11 milliseconds to obtain a single host (of which about 5 milliseconds is due to network communication and pseudo-device overhead), plus approximately 100 microseconds per additional host requested at the same time. It takes 2 milliseconds to confirm that a host is available; as mentioned earlier, this time could be reduced to a few hundred microseconds if the *select* call were modified not to require a remote procedure call. It takes 11 milliseconds to return a host explicitly via a pseudo-device communication, or 5 milliseconds to return outstanding hosts by closing the communication channel completely. These measurements are summarized in Table 7.1.

7.3.2 State Transfer

In Section 4.2, I described the mechanism used to transfer processes in Sprite. To summarize, migration consists of transferring a fixed amount of state plus a variable amount of state that depends on the migrating process. The state that must be transferred with all processes includes the process's current working directory and a number of fields from the process control block. The variable state includes open file streams as well as modified blocks for open files. It also includes virtual memory for processes that do not migrate at

Action	Time/Rate
Migrate null process	76 milliseconds
Transfer info for open files	9.4 milliseconds/file
Flush modified file blocks	480 Kbytes/second
Flush modified pages	660 Kbytes/second
Transfer *exec* arguments	480 Kbytes/second
Fork, *exec* null process with migration, wait for child to exit	81 milliseconds
Fork, *exec* null process locally, wait for child to exit	46 milliseconds

Table 7.2: *Costs associated with transferring processes.* All measurements were performed on SPARCstation 1 workstations. The time to migrate a process depends on how many open files the process has and how many modified blocks for those files are cached locally (these must be flushed to the server). If the migration is not done at *exec*-time, modified virtual memory pages must be flushed as well. If done at *exec*-time, the process's arguments and environment variables are transferred. The bandwidth of the RPC system is 480 Kbytes/second using a single channel, and 660 Kbytes/second using multiple RPC connections in parallel for the virtual memory system.

exec-time. At *exec*-time, the state includes the name of the file being *exec*ed, the arguments to the program, and the environment variables passed to the program.

Table 7.2 lists the costs associated with transferring processes. If a process had no open files, it would take about 100 milliseconds to select an idle host, start a trivial process on it, and wait for the process to exit.[1] In practice, the average time for *exec*-time migration is between 200 and 300 milliseconds, due to the extra time to transfer files and set up the process's stack with multiple command arguments and environment variables. (Refer to Chapter 8 for empirical data.) By comparison, it takes under 50 milliseconds to invoke a command locally. Note that even if the latency of remote invocation is five times as much as local invocation, it is quite small by comparison to the time needed to compile a small source file or run a simulation.

7.3.3 Forwarding Kernel Calls

After a process migrates away from its home machine, it may suffer from the overhead of forwarding kernel calls. The degradation due to remote execution depends on the ratio of location-dependent system calls to other operations, such as computation and file I/O. Figure 7.1 shows the total execution time to run several programs, listed in Table 7.3, both entirely locally and entirely on a single remote host. Applications that communicate

[1]Nearly all this time is processing overhead. Migration requires only three RPC's, two of which transfer little data and the third of which transfers only a few Kbytes. Starting a process requires extra RPC's to open the *exec*ed file, and an RPC is needed when the process exits. The remaining cost is due to processing.

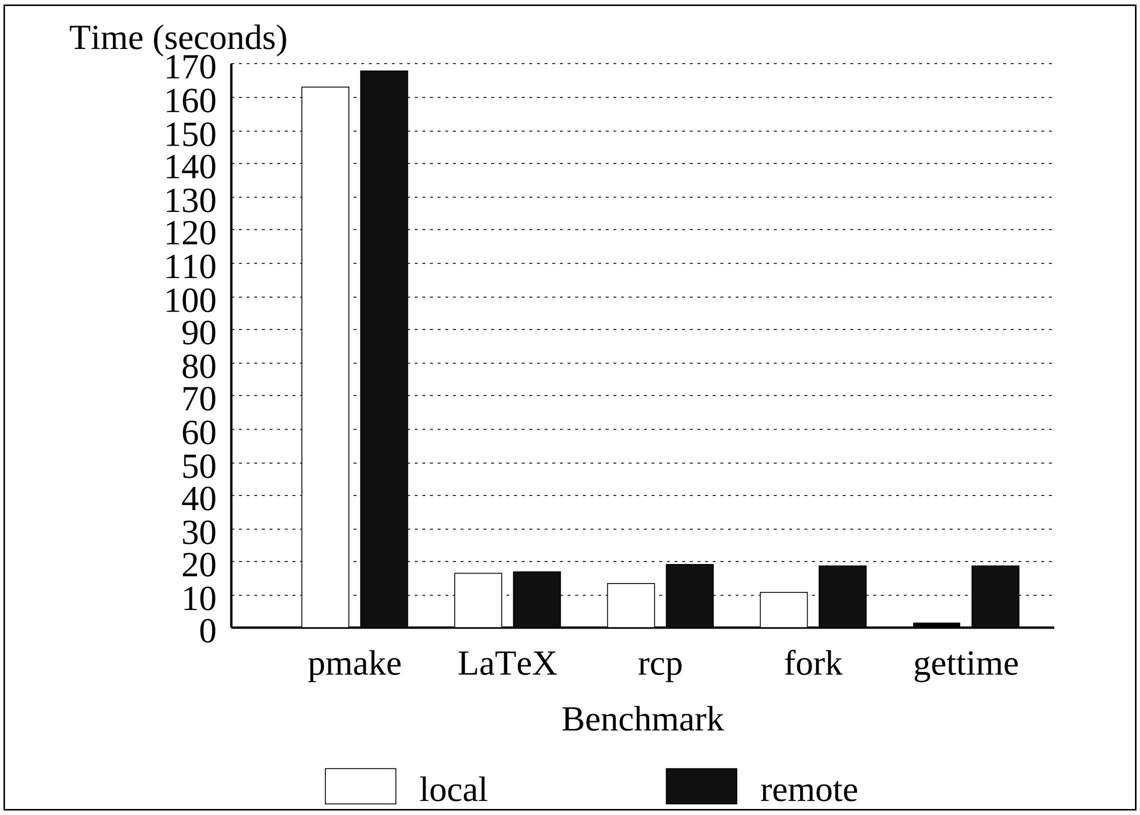

Figure 7.1: *Comparison between local and remote execution of programs.* The elapsed time to execute CPU-intensive and file-intensive applications such as *pmake* and LaTeX showed negligible effects from remote execution (3% and 1% degradation, respectively). Other applications suffered performance penalties ranging from 42% (*rcp*), to 73% (*fork*), to 3200% (*gettime*).

Name	Description
pmake	recompile *pmake* source on a single host using *pmake*
LaTeX	run LaTeX on a draft of a 15,000-line paper ([DO91])
rcp	copy a 1 Mbyte file to another host using TCP
fork	fork and wait for child, 1000 times
gettime	get the time of day 10000 times

Table 7.3: *Workload for comparisons between local and remote execution.*

frequently with the home machine suffered considerable degradation. Two of the benchmarks, *fork* and *gettime*, are contrived examples of the type of degradation a process might experience if it performed many location-dependent system calls without much user-level computation. The *rcp* benchmark is a more realistic example of the penalties processes can encounter: it copies data using TCP, and TCP operations are sent to a user-level TCP server on the home machine. Forwarding these TCP operations causes *rcp* to perform about 40% more slowly when run remotely than locally. As may be seen in Figure 7.1, however, applications such as compilations and text formatting show little degradation due to remote execution.

7.4 Application performance

The benchmarks in Section 7.3 measure the component costs of migration. This section evaluates the overall benefits of load sharing using migration. In Section 7.4.1 I present measurements of typical compilations using *pmake*. I demonstrate that parts of *pmake*'s execution are inherently sequential, limiting the benefits of parallel compilation. Despite this limitation, migration typically speeds up compilations by factors of three to six.

Section 7.4.2 discusses additional factors that limit the performance improvement of compilations using migration. Using several hosts, *pmake* can saturate the processors of both a Sun-4/280 file server and the SPARCstation 1 workstation running *pmake*, and is within a factor of two of saturating a 10 megabits/second Ethernet. Even considering only the parallelizable portion of *pmake*'s execution, using twelve hosts in parallel produced only a speedup of five. Over the entire compilation, *pmake* used processing time equivalent to three processors working at 100% capacity.

Section 7.4.3 compares CPU-intensive applications, such as simulations, to applications that interact heavily with the file system, such as compilations. The effective processor utilization of a sample set of 100 independent (parallelizable) simulations was over 800%, compared to 300% for the 12-way parallel compilation mentioned above.

7.4.1 Representative Pmake Performance

The *pmake* program, like *make* [Fel79], generates a dependency graph from its input specification, determines which files are out-of-date, and recreates each out-of-date file (or "target"). Unlike *make*, it can find disjoint dependency subgraphs and recreate independent targets in parallel. When an out-of-date target depends on multiple inputs, *pmake* waits until all of the inputs are up-to-date before executing the commands to create the target.

Figure 7.2 shows an example of *pmake*'s execution. A single *pmake* process parses dependencies and obtains the times at which each relevant file was last modified. In this example, *pmake* then creates two object files in parallel, and finally a single process uses the object files to link together a program. Obviously, the speedup that *pmake* can obtain due to load sharing depends in large part on the amount of work that can be done in parallel

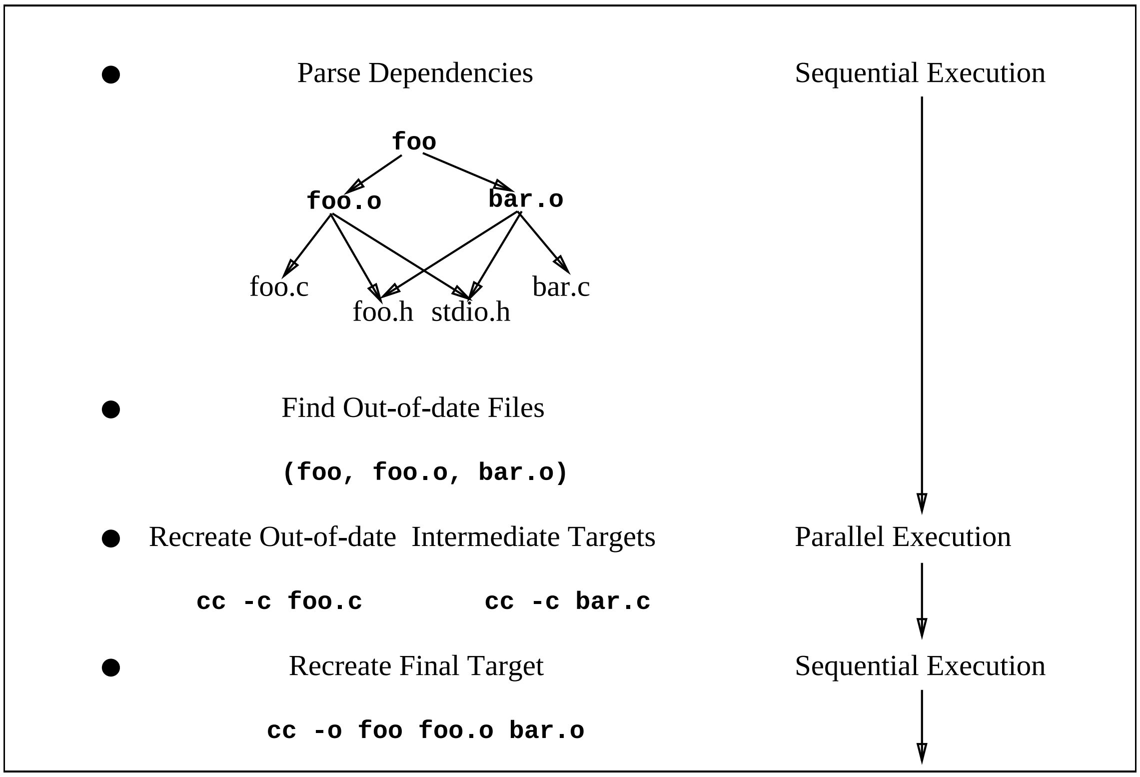

Figure 7.2: *Sample of* pmake *execution. Pmake* has one or more sequential phases, and one or more phases containing tasks that may be performed in parallel. In this example, *pmake* determines that `foo.o` and `bar.o` can be recreated in parallel, and then the two of them serve as inputs to the final step of the compilation.

by comparison to the work that is done by a single process. As "Amdahl's Law" suggests, by decreasing the time required for the parallelizable portion of *pmake*'s execution, the sequential portion eventually dominates the total execution time [Amd67]. Therefore, the greatest speedup is obtained when the sequential portion is a small fraction of the total execution time.

The performance improvement due to load sharing also depends on the ability of the application to execute efficiently on many machines simultaneously. On the one hand, *pmake* may be unable to keep processors busy; on the other, processors may become overutilized. Ideally, doubling the number of machines executing an application should halve the time needed to execute it. In practice, as the number of hosts increases beyond a small threshold, the marginal improvement from using additional hosts decreases.

To measure the performance of *pmake* with load sharing, I compare the time needed to compile different sets of files using a varying number of machines. The number of files in each set affects the maximum degree of parallelism attainable, while the number of machines can

Program Compiled	Number of Compiles	Number of Links	Avg. Lines per File	Lookups per Sec.	Sequential Time	Parallel Time	Speedup
pmake	49	3	433	62	162 s	55 s	2.95
kernel	276	1	550	98	1971	453	4.35
gremlin	24	1	435	43	180	41	4.43
TEX	36	1	514	24	259	48	5.42

Table 7.4: *Examples of* pmake *performance.* Sequential execution is done on a single host; parallel execution uses migration to execute up to 12 tasks in parallel. Each measurement gives the time to compile the indicated number of files and link the resulting object files together in one or more steps. Speedup is affected by inherent sequentiality that results from multiple link steps and checking dependencies, as well as by the amount of processing performed by each remote invocation. The amount of processing is partially determined by the code compiled by each remote invocation. Contention for the file server is greatly affected by the number of name lookups performed, as described below. The table lists the rate at which the file server performed name lookups during the sequential compilation, which is indicative of the contention it experiences during parallel compilation.

further constrain the parallelism allowed. In addition, by varying the number of machines used when many files must be compiled, it is possible to measure the marginal improvement from additional hosts.

Table 7.4 presents some examples of typical *pmake* speedups. These times are representative of the performance improvements seen in day-to-day use. Figure 7.3 shows the corresponding speedup curves for each set of compilations when the number of hosts used varies from 1 to 12. In each case, the marginal improvement of additional hosts decreases as more hosts are added. Table B.1 in Appendix B shows the detailed measurements of speedup as a function of the number of hosts.

The relatively low speedup obtained for compiling the source for *pmake* demonstrates the problem of keeping hosts busy. The program source is stored in a hierarchical structure consisting of two subdirectories within another directory. To recompile *pmake* from scratch, *pmake* recursively invokes two *pmake* processes sequentially, one per subdirectory. The files in each subdirectory are compiled and linked into a single object file. The processing for the second subdirectory cannot begin until the last file in the first subdirectory has been compiled and all the object files from the first subdirectory have been linked together. In the meantime, most of the hosts are idle.

7.4.2 Limiting Factors

The speedup factors obtained by the other benchmarks in Table 7.4 are determined by three problems: inherent sequentiality, file caching, and system bottlenecks. I analyzed the kernel

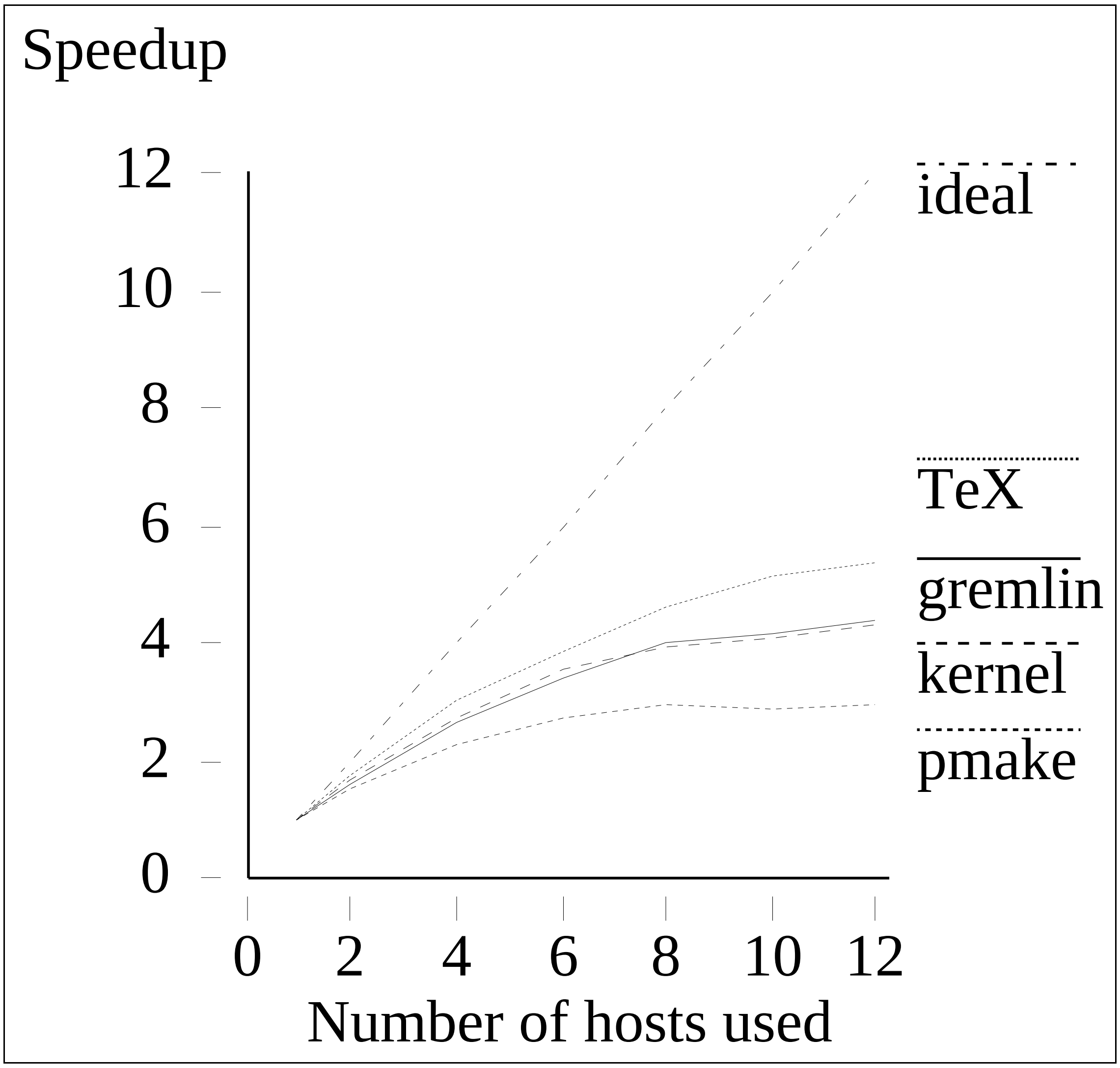

Figure 7.3: *Speedup of compilations using a variable number of hosts.* This graph shows the speedup relative to running *pmake* on one host (*i.e.*, without migration). The speedup obtained depends on the extent that hosts can be kept busy, and the amount of parallelization available to *pmake*. It also depends on system bottlenecks, as described below.

compilation, which is the largest benchmark in Table 7.4, to demonstrate these factors. In this benchmark, *pmake* determines the dependencies for 276 source files, compiles each file, and links the resulting object files into a single file. Figure 7.4 shows the total elapsed time to compile and link the files using a varying number of machines in parallel, as well as the performance improvement obtained.

In Figure 7.4, the "compile and link" benchmark includes the time to determine file dependencies and link object files together, which must be done on a single host; the "normalized compile" benchmark considers only the parallelizable portion of the execution. There are two reasons for the difference between the speedups obtained for the "normalized compile" benchmark and the "compile and link" benchmark: sequential execution and file caching. First, for this benchmark, *pmake* takes 19 seconds to determine file dependencies (including performing kernel calls to obtain the file attributes of every source file, header file, and object file), and 56 seconds to link the resulting object files together. Even if compiling the 276 source files were instantaneous, *pmake* could not execute faster than that without further changes to its structure.

Second, file caching can affect speedup significantly. As described above in Chapter 5, when a host opens a file for which another host is caching modified blocks, the host with the modified blocks transfers them to the server that stores the file. Thus, if *pmake* uses many hosts to compile different files in parallel, and then a single host links the resulting object files together, that host must wait for each of the other hosts to flush the object files they created. It then must obtain the object files from the server. In this case, linking the files together when they have all been created on a single host takes 56 seconds, but the link step takes 65–69 seconds when multiple hosts are used for the compilations. (As a more extreme example, in a similar benchmark compiling 139 source files on DECstation 3100 workstations, a link step increased from 8 seconds on a single host to 20–35 seconds using multiple hosts.)

System Bottlenecks

The speedup curves in Figure 7.3 and Figure 7.4(b) show that the marginal improvement from using additional hosts is significantly less than the processing power of the hosts would suggest. In the kernel benchmark, *pmake* is able to make effective use of about three-fourths of each host it uses up to a point (4-6 hosts), but it uses less than half the processing power available to it once additional hosts are used. The poor improvement is due to bottlenecks on both the file server and the workstation running *pmake*. Figure 7.5 shows the utilization of the processors on the file server and client workstation over 5-second intervals during the 12-way kernel *pmake*. It shows that the *pmake* process uses nearly 100% of a SPARCstation processor while it determines dependencies and starts to migrate processes to perform compilations. Then the Sun-4/280 file server's processor becomes a bottleneck as the 12 hosts performing compilations open files and write back cached object files. The network utilization, also shown in Figure 7.5, averaged around 20% and is thus not yet a problem. However, as the server and client processors get faster, the network may

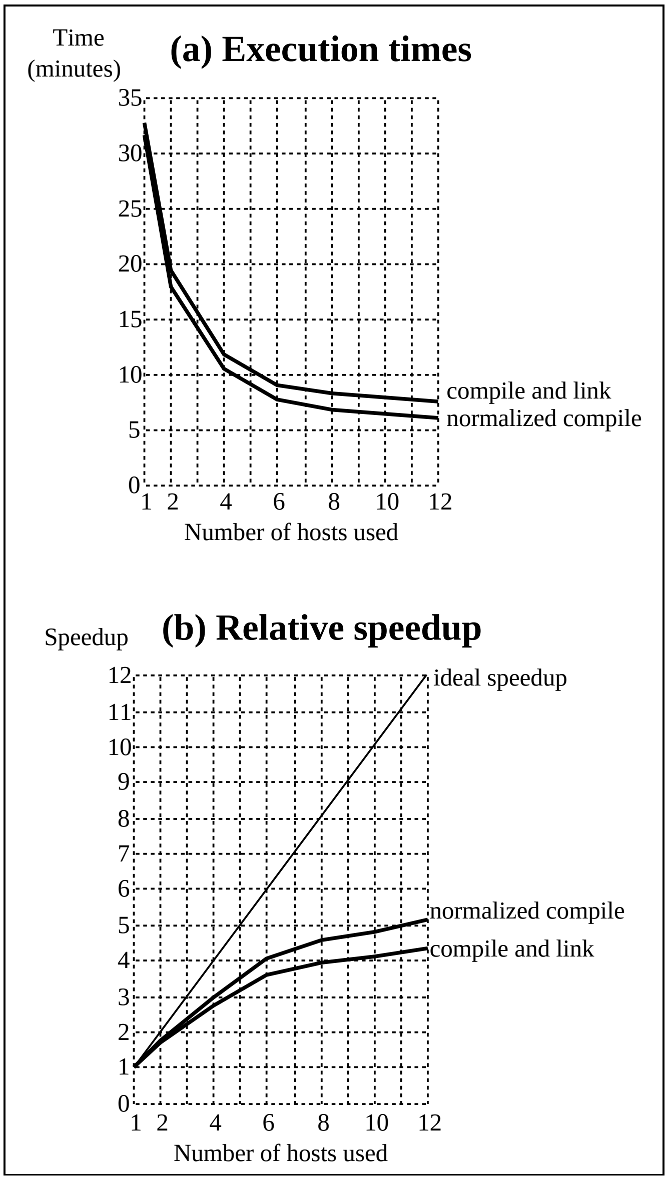

Figure 7.4: *Performance of recompiling the Sprite kernel using a varying number of hosts and the* pmake *program.* Graph (a) shows the time to compile all the input files and then link the resulting object files into a single file. In addition, it shows a "normalized" curve that shows the time taken for the compilation only, deducting as well the *pmake* startup overhead of 19 seconds to determine dependencies; this curve represents the parallelizable portion of the *pmake* benchmark. Graph (b) shows the speedup obtained for each point in (a), which is the ratio between the time taken on a single host and the time using multiple hosts in parallel.

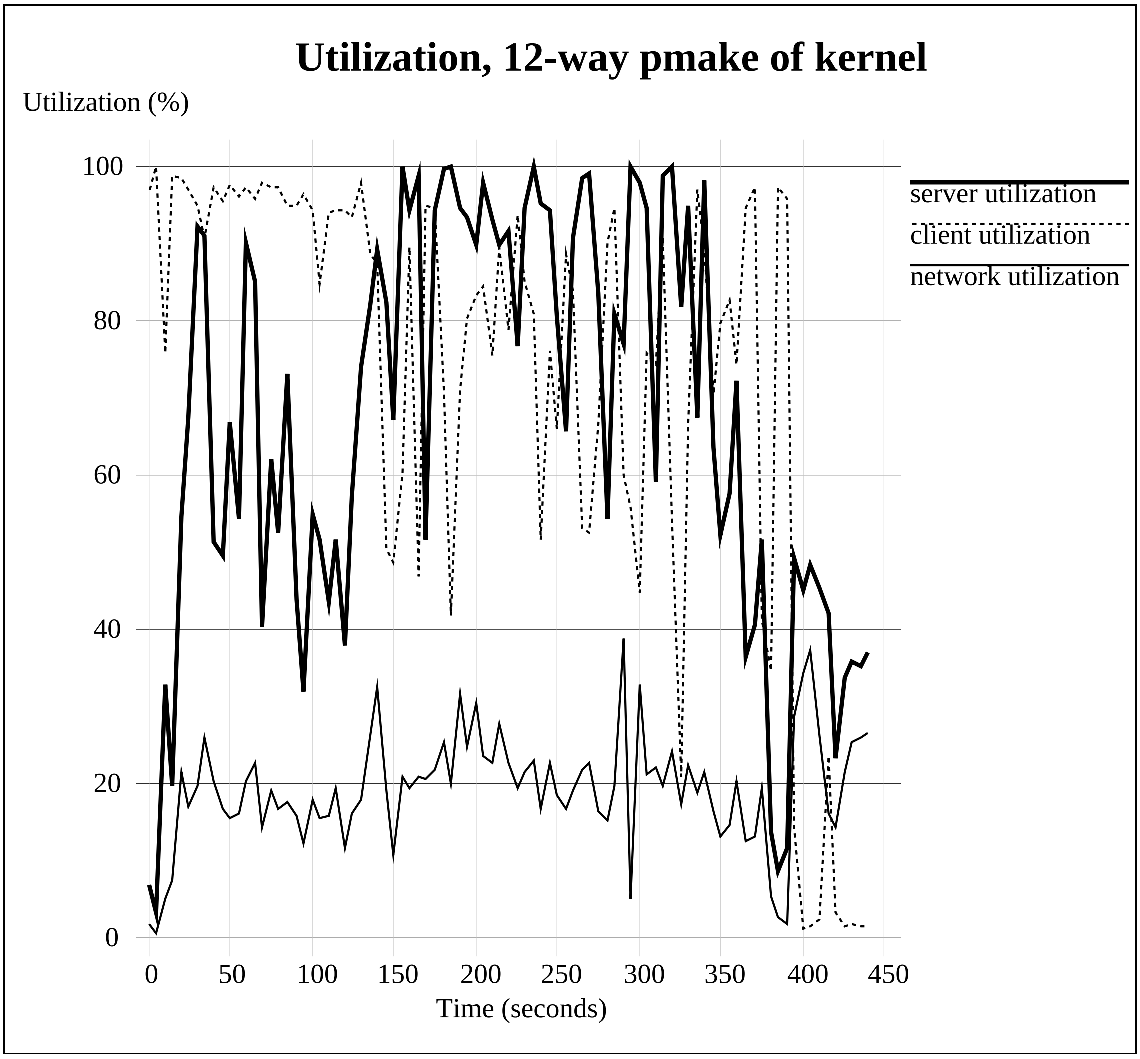

Figure 7.5: *Processor and network utilization during the 12-way* pmake. Both the file server and the client workstation running *pmake* were saturated.

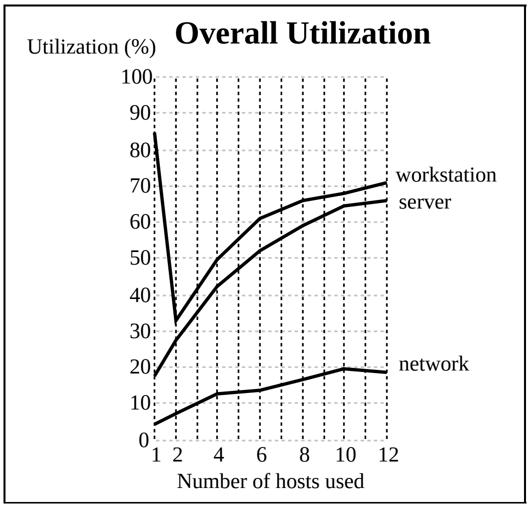

Figure 7.6: *Overall processor and network utilization as a function of hosts used.* When only a single host was used, it ran both the compilations and *pmake*, resulting in high utilization. When multiple hosts were used, the workstation running *pmake* did not run compilations itself.

---

easily become the next bottleneck. Figure 7.6 graphs the overall processor and network utilization for the kernel *pmake* as a function of the number of hosts, and Figure 7.7 plots server processor utilization versus time for the *pmake* runs using 1, 4, 8, and 12 hosts. The latter figure demonstrates that migration compresses the work performed by the file server into shorter and shorter intervals, thus making demands on the server (and the network) burstier.

The demand on the file server is due in large part to name lookups. In Sprite, client workstations send entire file names to file servers, and the file servers are responsible for traversing the directory hierarchy. File servers cache directory names, but they must still respond to each RPC that attempts to open or get the attributes of a file. Table 7.4 shows that the number of name cache accesses varies greatly from benchmark to benchmark. In particular, the file server handled nearly 200,000 name lookups during the course of the kernel compilation, which is an average of nearly 100 lookups per second when only one host is used. This rate is from 1.5 to 4 times the rate of lookups during the other benchmarks, and may be attributed to the large number of files included by each source file. Note that the *TEX* benchmark, which had the lowest rate of name lookups (as well as a high average number of lines per source file) obtained the highest speedup using migration.

7.4.3 Simulations

Though migration has been used in Sprite to perform compilations for nearly two years, it has only recently been used for more wide-ranging applications. Excluding compilations,

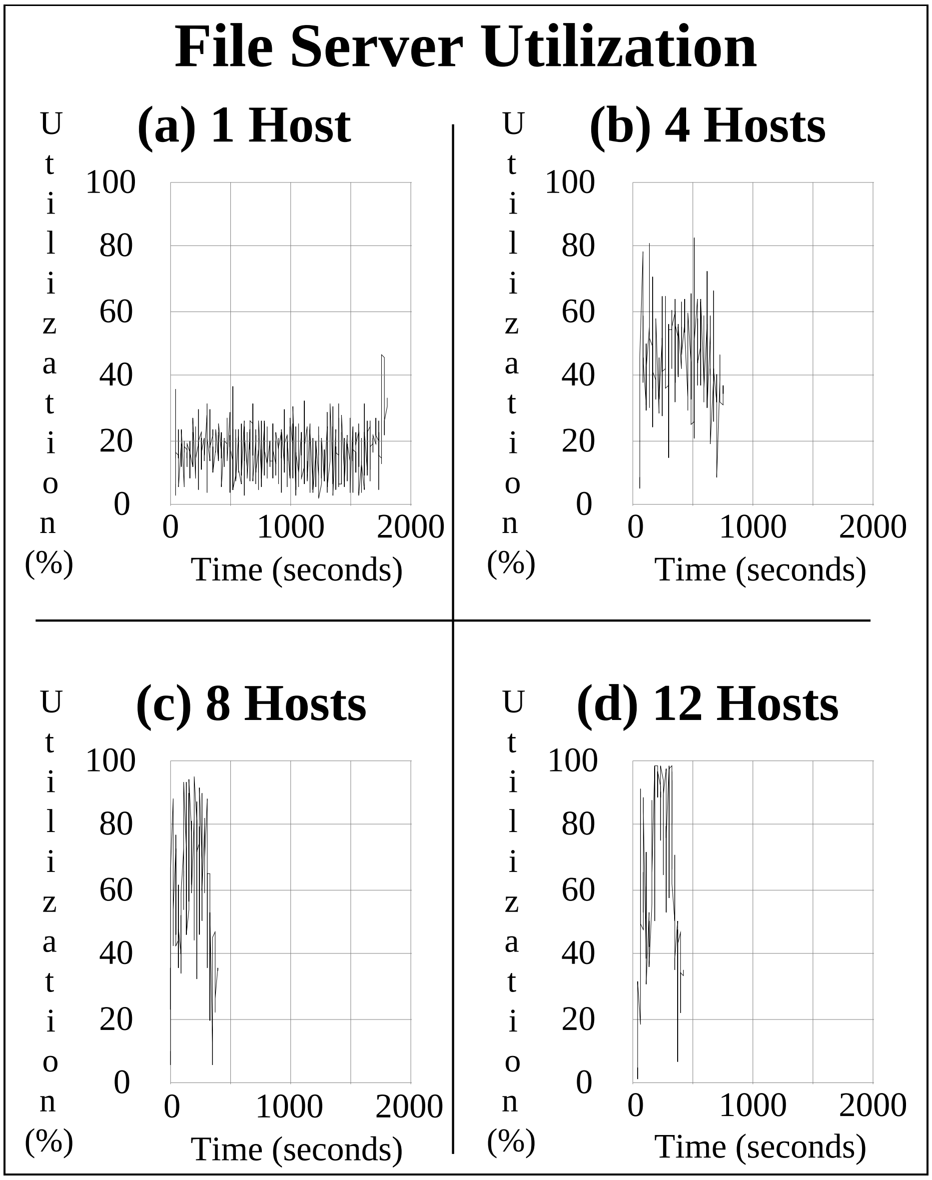

Figure 7.7: *Server processor utilization over time as a function of the number of hosts used in parallel.* The total processor time used on the server during the benchmark was roughly the same in each case, varying from 290–350 CPU seconds.

simulations are the primary application for Sprite's process migration facility. It is now common for users to use *pmake* to run up to one hundred simulations, letting *pmake* control the parallelism. The length and parallelism of simulations results in more frequent evictions than occur with most compilations, and *pmake* automatically remigrates or suspends processes subsequent to eviction.

In addition to having a longer average execution time, simulations also sometimes differ from compilations in their use of the file system. While some simulators are quite I/O intensive, others are completely limited by processor time. Because they perform minimal interaction with file servers and use little network bandwidth, they can scale better than parallel compilations do. One set of simulations recently obtained over 800% effective processor utilization—eight minutes of processing time per minute of elapsed time—over the course of an hour, using all idle hosts on the system (up to 10–15 hosts).

7.5 Conclusions

Process migration provides an efficient means to execute processes on multiple hosts simultaneously. The primary costs of migration are file transfer and, when processes do not migrate in conjunction with an *exec* call, virtual memory transfer. Other costs, such as host selection, may usually be amortized across multiple migrations.

The overall speedup of parallel applications in Sprite is limited primarily by contention for centralized resources such as file servers and, potentially, network bandwidth. *Pmake* is limited as well by the load it places on its host when it must interact with many child processes at once. Thus, compilations obtain speedup factors in the range of three to six, even when as many as 12 hosts are used at once to perform compilations. Long-running compute-bound applications such as simulations cause less contention and amortize the cost of process management over longer periods of time.

Because file data caching in Sprite is effective, name lookups are the greatest cause of contention for file server processing. In his thesis, Nelson estimated that adding client name caching would reduce file server utilization by as much as a factor of two [Nel88]. It would also decrease network utilization by up to a factor of two. Measurements of server load during parallel compilations demonstrate that name caching is imperative if the full benefits of migration are to be exploited. Using faster (or multiprocessor) file servers would also serve to increase the number of hosts that could execute an application in parallel before saturation became a problem.

Chapter 8

Empirical Results

8.1 Introduction

The measurements in the preceding chapter indicate that process migration can potentially improve the performance of a variety of applications significantly. The benefits of migration in practice, though, depend on more than just the speedup obtained by a particular compilation or the time it takes to migrate a minimal process. They depend upon the extent to which migration is used, the availability of hosts, the frequency and cost of eviction, and other characteristics of the environment. Since migration has been in regular use for over a year on a Sprite system with over 25 hosts and approximately 20 regular users, I have had the opportunity to gather empirical data on usage patterns. (Unless specified otherwise, measurements presented in this chapter were collected over a one-month period, from 3 August 1990 to 3 September 1990.) Though the measurements are not representative of all environments—for example, many of the users of Sprite are kernel developers—and the small scale of the system (in terms of both the number of users and number of hosts) may affect such factors as contention for idle hosts, these measurements can still validate some design decisions and suggest ways for improvement. (Further experience with migration in different environments—for example, 50–100 workstations running computer-aided design tools—would also be beneficial, but is not yet possible.)

Two aspects of migration were of particular interest while collecting measurements of migration. First, how much is migration used? If only a small fraction of all processing were performed using migration, one might wonder whether migration is needed at all. Second, how useful is migration by comparison to a simpler remote invocation facility? If evictions rarely occurred, or evicted processes used little processor time after eviction, then one might again wonder how useful migration is. In fact, as shown in this chapter, process migration provides substantial benefits with respect to both total processing capacity and eviction.

One way to estimate the benefit of remote execution is to measure the amount of processing performed by processes that are not executing on their home machine. In a sense, the amount of remote processing in the system does not represent an absolute increase in processing power: some processes might execute on remote hosts while their home machines remain relatively idle. However, measurements of the system show that parallel execution has provided an enormous improvement in processing capacity in practice. Hosts have executed applications that have collectively obtained as much as 7 seconds' worth of processing time per second of elapsed time, over a 3-hour period. Section 8.2 discusses the over-all benefits of process migration, as indicated by the amount of execution performed by remote processes.

The relative merits of migration and remote invocation are another issue entirely. For several years, the research community has debated the desirability of process migration with respect to load sharing. Eager, *et al.*, reported that process placement is much more important than migrating active processes and that migration provides limited improvement beyond placement [ELZ88]. Krueger and Livny, on the other hand, found that migration *can* provide significant improvement beyond initial placement [KL88]. Finally, Cabrera measured actual process lifetimes on a collection of VAXes running UNIX and found that nearly all processes were short (median lifetime of 0.4 seconds, with more than 78% of processes living less than one second) and that processes that live a long time are expected to live longer; he concluded that process placement at creation time would be wasteful, and instead long-running processes should be migrated after they have executed for a period of time [Cab86].

I would argue that migration for the purpose of ensuring workstation autonomy is at least as important as migration for redistributing load. In our environment, in fact, these two goals are interrelated. When a user returns to a workstation with foreign processes, Sprite migrates those processes to their home machine. From Cabrera's study, one might question whether those processes are likely to execute for such a short period of time that migrating them is unnecessary. Section 8.3 reports on the characteristics of process eviction in Sprite. Although many processes terminate quickly after eviction, a small number of processes perform a substantial amount of processing after being evicted. For this reason, *pmake* automatically remigrates evicted processes to another host. Remigration avoids overloading the home machine with multiple CPU-bound evicted processes.

In addition to measurements of remote execution and eviction usage, I measured the typical frequency and costs of typical operations, such as the time to invoke a program on a remote host. These measurements, reported in Section 8.4, demonstrate the additional cost that open files and modified file blocks incur in practice. Section 8.4 also reports on other aspects of migration, such as the interaction between migration and cache consistency.

Section 8.5 discusses Sprite's host selection facility. It reports statistics such as the rate of requests for idle hosts, the availability of idle hosts, and the rate at which hosts are reclaimed.

Section 8.6 concludes the chapter.

Host	Total CPU Time	Remote CPU Time	Fraction Remote
garlic	314,218 secs	228,641 secs	72.77 %
crackle	172,355	14,451	8.38 %
sassafras	158,515	138,821	87.58 %
burble	151,117	2,352	1.56 %
vagrancy	107,853	81,343	75.42 %
buzz	96,402	260	0.27 %
sage	92,063	32,525	35.33 %
kvetching	91,611	26,765	29.22 %
jaywalk	75,394	24,017	31.86 %
joyride	58,231	6,233	10.70 %
Others	857,532	120,727	14.1 %
Total	2,175,291	676,135	31.08 %

Table 8.1: *Remote processing use over a one-month period.* The ten hosts with the greatest total processor usage are shown individually. Sprite hosts performed roughly 30% of user activity using process migration. The standard deviation of the fraction of remote use was 25%.

8.2 Overall Usage

In order to gather usage measurements, I modified the operating system to record information about the time accumulated by processes. When a process exits, the total processor time used by the process is added to a global counter; if the process had been executing remotely, its time is added to a separate counter as well. (These counters therefore exclude some long-running processes that do not exit before a host reboots; however, these processes are daemons, display servers, and other processes that would normally be unsuitable for migration.) Over the measured interval, remote processes accounted for about 30% of all processing done on Sprite. Some hosts ran applications that made much greater use of remote execution, executing as much as 70–90% of their processor cycles on other hosts. Table 8.1 lists some sample processor usage over this period.

Table 8.1 lists usage over a one-month period, but the benefits of migration are more apparent if one considers shorter time-frames as well. For example, the host with the greatest overall processor use ("garlic") had a three-hour period during which it obtained an average of over three seconds of remote processor time per second of elapsed time, in addition to one second of local processor time. "Sassafras" had a period when it obtained seven seconds per real second. Both of these hosts were used for long-running parallel simulations several times during the month.

As a final note, during the measured one-month period the total processor utilization of the system was 2.3% (averaged over all times of day). Its utilization is an order of magnitude smaller than the utilization reported for Condor [LLM88], but as new long-running parallelizable applications execute on Sprite, its utilization will likely increase significantly.

8.3 Process Eviction

When a user returns to a workstation, the execution of foreign processes on the workstation may degrade the user's interactive response. Sprite uses eviction to prevent long-term performance degradation from foreign processes, but other alternatives are possible. If users rarely returned when foreign processes executed on their host, then the system could choose not to provide any special support for ownership. Measurements of our current environment indicate that users rarely find processes on their hosts, but evictions occur often enough to warrant having a means to deal with foreign processes. (See Section 8.5.3 for more information). Also, if processes only executed a short time subsequent to eviction, they could be permitted to complete on their remote host.

I measured the processor time used by processes after they had been evicted. This time indicates how long a process would continue to execute on the remote host in the absence of process migration, unless it were terminated. (For this reason, a process that was evicted multiple times would be measured from its first eviction.) I also measured the average time to evict processes and the average number of processes evicted each time a user reclaimed a host.

Though compilations tend to execute for a short enough time that eviction might be unnecessary, some processes execute a substantial time. Not surprisingly, the hosts that used the most remote processor time executed processes for substantial periods of time following eviction. For example, a total of 82 processes that were evicted to "sassafras" executed for an average of 21 minutes after they were evicted. Two other hosts, "garlic" and "vagrancy," had a total of 250 processes evicted, each executing an average of 5 minutes following eviction. Most other hosts had very few processes evicted, and those processes executed for at most a few seconds after eviction. One may conclude that eviction (and subsequent remigration to another idle host) is useful for long-running processes such as simulations, and much less important for transient applications such as compilations.

Another metric that affects the feasibility and desirability of eviction is the total time needed to evict processes. When a user types a keystroke, the system immediately starts to evict foreign processes and relinquish their resources, such as memory. An average of 3.26 processes were evicted each time an eviction occurred. The average time to relinquish a host completely was 3.44 seconds, with a standard deviation of 3.65 seconds across different hosts. The cost of eviction is of course a function of the processes evicted, and processes with many modified memory pages or file blocks will take longer than average to evict. The next section reports average costs associated with migration (evictions and otherwise).

Type	Count	Fraction	Number/Hour/Host
Exec-time	46382	86 %	1.76
Eviction	3037	6 %	0.12
Voluntary migration remote ⇒ home	3431	6 %	0.13
Voluntary migration home ⇒ remote	972	2 %	0.04
Total	53822	100 %	2.05

Table 8.2: *Frequency of different forms of migration over a 1-month period.* Nearly all migration has occurred in conjunction with the *exec* kernel call, with the remaining migrations roughly evenly divided between eviction and voluntary migration home of long-running processes. Processes rarely migrate away from the home machine except at *exec*-time. On average, a total of 2 migrations per host per hour occur.

8.4 Migration Frequency and Costs

Process migration occurs in four ways in Sprite:

- *Exec*-time migration, when a process changes hosts at the same time it replaces its address space with a new execution image. *Exec*-time migration is the simplest form of migration, because the kernel transfers less state than when a process migrates during execution.
- Eviction, when a process moves from a remote host back to its home machine as a result of someone reclaiming the remote host. This happens when a user returns or when the centralized server reclaims a host to assign it to another process.
- Voluntary migration from a process's remote host to its home machine, transferring the entire address space. In Sprite, when a process on a remote host migrates to another remote host, it must migrate home first. (This simplifies the protocol for migration, since the home machine is involved any time a process belonging to it migrates.) Migration between two remote hosts normally occurs when a foreign process starts a program that itself performs remote invocation—for example, if *pmake* runs another *pmake* on a remote host and the latter *pmake* executes tasks in parallel.
- Voluntary migration from a process's home machine to a remote host (transferring the entire address space). This form of migration is usually a result of *pmake* remigrating a process to an idle host after it has been evicted from another host.

Table 8.2 lists the relative frequency of each form of migration. *Exec*-time migration is the most common case, constituting 86% of all migrations. Eviction and voluntary migration are both relatively infrequent; they are considered separately because their function is

Measurement	Average Rate	Standard Deviation Among Hosts
Number of open files	4.53/migration	1.00
Number of modified file blocks	0.95/migration	0.28
Number of modified VM pages	36.62/migration	6.50
Size of migration RPC buffer	6.95 Kbytes/migration	0.82
exec-time migration	0.33 sec/migration	0.43 secs
eviction	2.98 sec/migration	3.11 secs
other migrations	0.95 sec/migration	0.16

Table 8.3: *Characteristics of migrating processes.* The time to migrate a process depends on several factors, including the number of open files the process has, how many modified blocks for those files are cached locally, the number of modified virtual memory pages the process has (if migration is not done at *exec*-time), and the total size of the buffer used to transfer the process's state to the target machine. Measurements were taken on 15 SPARCstation 1 workstations over the course of one month. (Note that the workstations were running kernels with tracing of remote procedure calls and some other events enabled, so performance is not comparable to the measurements of minimal overhead presented in the last chapter.)

different. When evicting a foreign process, the goal is to relinquish its resources as quickly as possible and to impact the user on the remote host as little as possible. When a process migrates voluntarily, however, the purpose of migration is to make the process execute more quickly, and there is no need to relinquish resources immediately. Although Sprite uses the same paradigm in both cases, it would be possible to distinguish between the two cases: for example, performing lazy copying of virtual memory when processes migrate voluntarily, but flushing VM to a server when evicting processes.

Table 8.3 lists statistics about the cost of migration in Sprite. Since migration is currently used primarily for parallel compilations, most migrations involve four open files: the three "standard input, output, and error" streams, and an input file with commands for a shell. The input file, which is typically shorter than 1 Kbyte, must also be flushed to the file server. Table 8.3 shows as well that the cost of eviction is significantly more than that of *exec*-time migration, but it is still small enough to impose minimal overhead on the returning user.

File cache consistency is another aspect of migration overhead. In Section 5.2.2, I described four scenarios for the state of a file as a result of migration. Empirically, more than half of all transferred files change from being cacheable to being uncacheable. This change is because the primary client of migration is *pmake*, and it uses pipes and a command

Cacheability		Count	Fraction
Before Migration	After Migration		
read-only, cacheable	(same)	12,650	18 %
writable, cacheable	(same)	1,236	2 %
uncacheable	(same)	13,478	19 %
uncacheable	cacheable	2,759	4 %
cacheable	uncacheable	39,939	57 %
Total		70,062	100 %

Table 8.4: *Caching behavior as a result of migration.* Most files change from being cacheable to being uncacheable due to migration, but other patterns are also common.

script to communicate with its child processes. The pipes are by necessity shared between *pmake* and its children. A script is created for each child and is executed after migration. The parent *pmake* does not close its stream into the script, so streams to it exist on multiple hosts after migration.[1] Other files that are open during migrations are typically either accessed in a read-only manner by all processes or are uncacheable before migration begins. Table 8.4 summarizes these measurements of cache consistency.

8.5 Host Selection

In Chapter 6, I referred to several assumptions that guided the design of Sprite's host selection facility: the wide-spread availability of idle hosts, the correlation between idle time and time to eviction, and the clustering of requests for hosts. In order to validate those assumptions, I instrumented the current Sprite host selection facility, recording usage patterns in a collection of about 28 Sprite SPARCstation 1 and DECstation 3100 workstations over a 25-day period. (This count does not include file servers, which do not participate in load sharing, or Sun-3 workstations, which run Sprite but are not used much.) Since migration in Sprite is used only among homogeneous machines, the host selection facility assigned hosts from within two pools averaging 14 hosts apiece; thus the maximum number of hosts available to any one process was 14 rather than 28.

I measured host selection in order to answer a number of questions that relate to the assumptions given above:

- How often are hosts idle?

[1] This is apparently a bug in *pmake* and is not necessary for normal operation. It has the effect of skewing the distribution toward uncacheable files, but for each script that could be cached, there exists a pipe that could not be. Therefore, half of the 57% of files that become uncachable could possibly be cached after migration.

Time Frame	13 months		25 days	
	In Use	Idle	In Use	Idle
weekdays	31%	66%	29%	71%
off-hours	20	78	26	74
total	23	75	27	73

Table 8.5: *Host availability.* Measurements are presented for both the 13-month interval for which statistics were gathered and the 25-day period during which the host selection facility was instrumented. Weekdays are Monday through Friday from 9:00 A.M. to 5:00 P.M. Off-hours are all other times.

- How many hosts does a process typically request during its lifetime, and how often do processes obtain as many hosts as they request?
- What is the relationship between evictions and idle time?
- How often are hosts reclaimed due to fairness considerations?
- How often are hosts reused by the same user?

8.5.1 Availability of Idle Hosts

One metric that affects the usability of a load sharing facility is the fraction of hosts available at any given time. Over the course of the past 13 months, I have periodically recorded the state of every host (active or idle) in a log file. In this time, a surprisingly large number of hosts have been available for migration at any time, even during the day on weekdays. This is partly due to our environment, in which several users own both a Sun and a DECstation and use only one or the other at a time. Some workstations are available for public use and are not used on a regular basis. However, after discounting for extra workstations, I still find a sizable fraction of hosts available, concurring with Theimer [TL88], Nichols [Nic87], and others. Table 8.5 summarizes the availability of hosts in Sprite over the 13-month interval, as well as the 25-day period during which the host selection facility was instrumented. (The measurements over the shorter period are presented for consistency with later measurements of the host selection facility.)

8.5.2 Host Allocations

I measured the success rate of host selection by recording the maximum number of hosts requested by each process and the number of hosts obtained. As a rule the number requested is either the number obtained or one greater than the number obtained, because a client of the host selection facility tends to request each host individually and will only request host N after it has successfully obtained $(N - 1)$ hosts. (This unfortunately means that there is no way to determine how many hosts could ultimately have been used by a process that has been denied a host.) During the 25-day period for which measurements were collected, 17,800 processes made a total of 134,000 requests for idle hosts (for a rate of 3.25 requests/minute). Considering only the maximum number of hosts ever requested by each process, a total of 46,400 hosts were requested. Over 15,000 of the processes obtained as many hosts as they wanted (an 86% success ratio). The maximum number of hosts requested by these processes averaged 2.6 hosts, with a maximum request of 16 hosts and a standard deviation of 4.58 hosts. They obtained an average of 1.8 hosts, with a standard deviation of 3.41 hosts. Figure 8.1 shows the distribution of hosts requested and hosts obtained.

8.5.3 Host Idle Times

Each time a host becomes active after being idle, it may evict processes. If idle hosts are in use for migration much of the time, then the likelihood of eviction will be high regardless of the policy for assigning hosts to processes. On the other hand, if most hosts are not in use at any given time, then evictions will only occur if processes happen to be executing on a host when it becomes active again. In an environment such as ours—with an average availability rate of 70–80%—the system should select hosts that are unlikely to become active soon.

Idle time appears to be an important factor in making the determination of when a host will become active again. I measured the amount of time hosts were idle when they were selected for remote execution and the amount of time they were idle when they became non-idle. Generally, hosts that were idle for at least 30 seconds remained idle for a significant length of time (averaging 26 minutes). Hosts were used by applications for an average of only 62 seconds before being returned to the pool of idle hosts. Therefore, even if the cost of eviction were substantial, evictions would be infrequent enough that it would be profitable to use hosts that were idle only 30 seconds or a minute. In the period under consideration, only 700 of the 130,000 host assignments were revoked because a host became active. The average idle time of all hosts assigned was 17 hours, but the average idle time of those hosts that later evicted foreign processes was only 4 minutes. Thus, the policy of assigning hosts in order of idle time is justified: since most evictions occur on hosts that have only been idle a short time, even though most processes are allocated hosts that have been idle a long time, the likelihood of eviction is high only when hosts that have just become idle are assigned. Table 8.6 summarizes these results.

8.5.4 Fairness Considerations

In Chapter 6, I described how a host selection facility may choose to reclaim hosts from one process to assign them to another. In Sprite, hosts are reclaimed due to fairness about as often as they are reclaimed because a workstation owner returns: during the measured

Measurement	Value
Average idle time of hosts before becoming active	26 minutes
Average time host used before being returned to pool	62 seconds
Average idle time of hosts when assigned	17 hours
Average idle time of hosts that evicted processes	4 minutes
Number of evictions/allocations	700/130,000

Table 8.6: *Relationship between idle time and eviction likelihood.* Only when processes migrate onto hosts that have been idle a short time is eviction likely.

period, hosts were reclaimed due to fairness 760 times and due to eviction 700 times. This works out to roughly an 8% likelihood that a process will lose at least one host due to eviction or fairness.

8.5.5 Reusing Hosts

In order to find out how effective Sprite's simple host assignment algorithm is with respect to reusing idle hosts, therefore potentially running with warm caches, I recorded the number of times that hosts were assigned to processes on the same host twice in a row. I excluded gaps caused by a host becoming active, since in those cases the host's cache would presumably be loaded with other data.

Interestingly, the rate of host reuse has changed dramatically over time. In an earlier measurement period, hosts were assigned to the same requesting host twice in a row in 93% of all assignments. This trend was primarily due to the fact that host requests were bunched in groups of requests from the same host, so processes on the same host would obtain and release the same set of hosts repeatedly before a process from another host requested any hosts. In fact, considering every pair of successive host assignments in isolation (and only considering a single machine architecture at a time), 95% of successive requests were from the same process or from two processes on the same host. In the measurement period described in this chapter, however, only 36% of requests came from the same host as the preceding request. Because requests were no longer bunched together, hosts were more likely to be assigned to processes on different hosts each time they became available. Only 49% of all assignments went to processes on the same host twice in a row, compared to the earlier figure of 93%. The different between the two measurements is due to a change in workload: the first measurement took place when migration was used almost exclusively for short-lived applications such as compilations, while in the second measurement period multiple long-running simulations contended with compilations for idle hosts.

Since the simple approach to assigning hosts in order of greatest idle time does not take caching into account, one might wish to assign hosts on the basis of the last process that used them. However, it is not clear that the benefits of warm file caches outweigh the benefits of a decreased likelihood of eviction. If caching were more important, then it would be appropriate to modify the Sprite host selection facility to record recent host assignments and attempt to reassign hosts to the same client hosts repeatedly. Given the effectiveness of the current algorithm with respect to avoiding evictions, the system should be careful to consider caching effects only within a set of equally desirable candidates for migration—for example, hosts that have all been idle for a considerable period of time.

8.6 Conclusions

The measurements of process migration usage in Sprite presented in this chapter suggest three conclusions: first, that in actual use the overhead of migration is low relative to the processing demands of applications that use it; second, that migration in Sprite is used a significant amount and has the capacity to be used significantly more; and third, that evictions are rare but occur often enough to justify providing migration as a mechanism for reclaiming hosts.

As was described in Chapters 4 and 7, the cost of migration depends on the size of its address space (if it is migrating at some time other than as part of an *exec* call), the number of dirty file blocks it has written, and the number of open files a process has. In practice, *exec*-time migration—the most common type of migration—usually involves very few open files and dirty file blocks and no virtual memory, so it takes a few hundred milliseconds to complete. Eviction is more costly, but even typical evictions take only a few seconds at most. Therefore, for the applications using migration thus far, the benefits of increased processor utilization (reported in Section 8.2) far outweigh the potential cost of occasional evictions.

Over a 25-day interval, remote processor usage accounted for 30% of all processing, while 73% of the system on average could be executing remote processes. Thus, the migration facility is being used a considerable amount, though it has the capability to provide even more processing capacity. Idle hosts were sufficiently available to satisfy 86% of all processes that use the load sharing facility. Note that as more long-running parallel applications use the facility, the likelihood of contention for idle hosts will increase, but the total processing performed using migration will also increase as the system utilization rises.

Finally, the low rate of evictions in Sprite might suggest that migration at arbitrary times is unnecessary, and the implementation complexity of full process migration might be avoided. However, the low rate of evictions in Sprite results from the low over-all utilization of idle hosts, combined with the policy that gives preference to hosts that have been idle the longest. Since machines often are idle for relatively short periods of time, if there were applications to make use of idle hosts much more of the time, migration would make it possible to take advantage of short available intervals without the need to checkpoint and restart applications manually. I am starting to see this behavior to a limited extent as Sprite users execute multiple instances of long-running simulations using migration, and I expect such applications to become more common as load sharing facilities become more sophisticated.

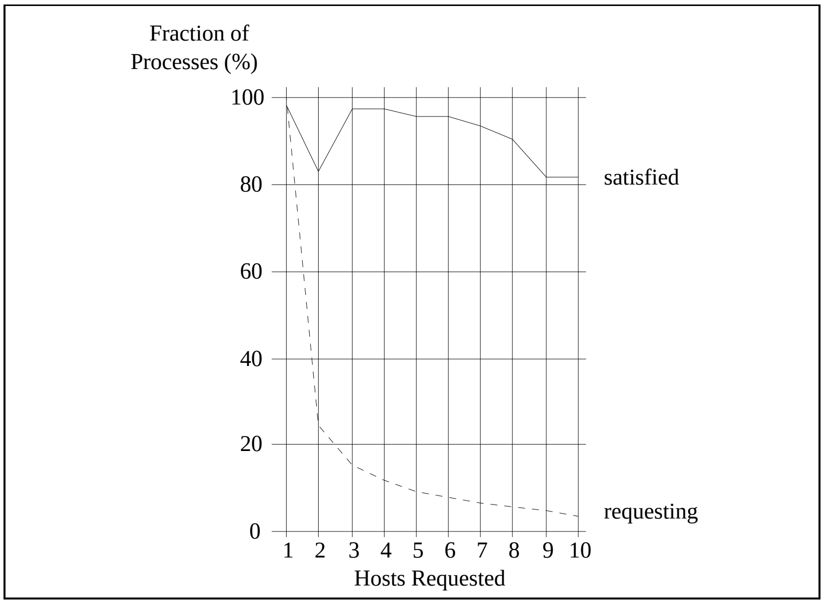

Figure 8.1: *Distribution of host requests and satisfaction rates.* For a given number of hosts, shown on the X-axis, the line labeled *requesting* shows the fraction of processes, using the host selection facility, requesting at least that many additional hosts. The line labeled *satisfied* shows, out of those processes that requested at least that many hosts, the fraction of processes that successfully obtained that many hosts at one point during their execution. Thus, 98% of all processes were able to obtain at least one host, and over 80% of processes that requested at least ten hosts obtained 10 hosts. Only 24% of processes requested more than one host. The sharp decrease in the satisfaction ratio of processes that requested exactly two hosts is attributable to fairness considerations. At times when few hosts are available, nearly any process that requests one host can obtain it, because it will be taken from a process with two or more hosts. A process that requests a second host will receive it only if one is free or another process is already using three hosts. Processes that requested three or more hosts had already received a second host, so contention for idle hosts during their execution was not as much of a problem.

Chapter 9

Conclusions and Future Work

9.1 Introduction

In this thesis I have presented the design and implementation of a transparent process migration facility in the context of the Sprite operating system. Sprite uses migration to invoke commands on remote hosts and to evict foreign processes when hosts are reclaimed. The degree of transparency provided by Sprite ensures that migration does not affect the interaction of a migrated process with other processes or the user who invoked it. The facility has been in regular use for nearly two years, which has allowed me to measure its use empirically.

9.2 Summary of Results

The most important result of this thesis is its demonstration of the effectiveness of process migration in a production environment. Migration has provided a significant increase in available processor capacity, accounting for over 30% of all processing performed on Sprite over a one-month interval. Over shorter periods of time (*e.g.*, hours), applications have used migration to obtain the processing equivalent of up to seven processors on a single (uniprocessor) host. Compilations and other file-intensive applications obtain less speedup because of contention for the processor on Sprite's file server, as well as high load on the workstation controlling the compilation. In those cases, speedups in the range of three to six are typical.

To determine whether the use of eviction is warranted, I measured the processing demands of evicted processes. I found that the execution time subsequent to eviction is bi-modal, with many processes executing a short time after eviction and some processes executing at length. Migration as a means to reclaim hosts might be over-kill in an environment that executed only short-lived applications, but long-running CPU-bound applications such as simulations need to be migrated or terminated if foreign processes are not to degrade interactive performance.

I have described the design of Sprite's migration facility, including the methods used to ensure transparent remote execution. Sprite associates a *home machine* with each process in order to make process migration transparent to both processes and users. Processes appear to execute on their home machine throughout their lifetime, regardless of where they physically execute. If they migrate away from their home machine, they depend on that host for resources, but they have no dependencies on a remote host after migrating back home.

Finally, I have considered several criteria and possible implementations of host selection for an environment with a shared file system and point-to-point communication. Several architectures are possible for selecting hosts for load sharing, including a shared file database, a central host selection server, and distributed host selection servers. Each of these provides acceptable performance and can scale to a sufficient number of hosts to be suitable for a system such as Sprite. However, a centralized host selection facility is the only one that provides high performance while ensuring that host assignments are fair and reflect the most current state of the system.

Empirical measurements have demonstrated a correlation between the amount of time a host is idle and the time before it will become active again. For this reason, among idle hosts, hosts are assigned in order of decreasing idle time. To avoid flooding a host with multiple foreign processes, the host selection facility assigns at most one process to each host at a time. These two simplifications have proven effective in practice, keeping the eviction rate at less than one eviction per 100 migrations.

9.3 Future Work

So far, the Sprite community has used process migration for a limited number of tasks in a community of tens of users and workstations. This research could be extended in any number of ways, but the most immediate improvements might be to increase both the degree to which load sharing occurs and the scale of the system measured. In addition, some basic changes to the functionality of migration would make the facility more useful.

The easiest way to increase the frequency of load sharing would be to increase the number of parallel applications using the system. Currently, *pmake* is the only parallel application that most users execute with any frequency, and most invocations of *pmake* are to compile files. Some users now use *pmake* to perform parallel execution of a single simulator with varying parameters, but such long-running applications are still rare: the total utilization of the system over a one-month period was only 2%. Other applications are likely suitable for migration, so as parallel applications for multiprocessors become more common, those applications that do not rely on shared memory should be appropriate for process migration as well. Also, traditionally sequential operations in existing applications may be parallelizable: for example, the **foreach** loop in *csh* could easily be executed in parallel if its semantics were redefined not to execute commands in a particular order.

Increasing the scale of the system has ramifications beyond those relating to load sharing. For example, in his thesis, Welch discussed the issue of Sprite file servers handling ten times as many clients as they currently do [Wel90]. It would be edifying to expand Sprite to handle many more machines, and to evaluate how the file system and host selection facility are stressed. Obviously, the file system is already stressed by some applications of migration, but performing file name lookups on client workstations should reduce contention considerably. (Gray and Cheriton's file "leases" [GC89] are promising in this respect.) With many hosts the host selection facility may also potentially become a bottleneck, unless host assignments may be cached effectively to reduce the rate of requests to a central server.

Finally, the existing process migration facility needs additional functionality in the following respects:

- There should be more automatic management of migration, especially in the case of eviction. Currently, only *pmake* provides support for remigrating or suspending processes after eviction. A system-wide facility should manage remigration. The system could also potentially detect when more long-running processes than processors are executing on a host, and migrate processes to distribute load.

- In the absence of network-wide shared memory, processes that share memory should be permitted to migrate together to the same host (currently, they cannot migrate at all). Even better, the system could permit processes on different hosts to share memory.

- The system should permit remote execution on machines of different types. Currently, migration is supported only among homogeneous architectures, even when the execution image of a process changes during migration as the result of an *exec* kernel call. The reason for this restriction is that kernel calls are forwarded to the home machine, and no allowance has yet been made for forwarding calls between machines of different byte-orders. However, there is no fundamental reason for keeping this restriction except a lack of time to modify the implementation—LOCUS, for example, supports heterogeneous remote invocation [PW85]—and allowing heterogeneous remote invocation would expand the pool of available hosts for remote invocation.

- The system should also permit a nontransparent variant of migration, in which there would be no residual dependencies on the home machine—in fact, the process's home machine itself would change to a different machine. This extension would permit processes to move in order to avoid a machine that is about to be shut down. It would also permit commands to be executed in the environment of another machine, similar to *rsh* but more efficient and permitting files to be inherited.

- Host selection in a multiprocessor environment must be evaluated. The paradigm in Sprite for reserving access to a host has been effective on uniprocessors, but with multiple processors available per host, different techniques may be more appropriate.

9.4 Conclusion

Though remote invocation is common in distributed systems, process migration has been implemented in only a few. However, existing implementations of migration prove that migration is feasible, and practical experience with migration in Sprite demonstrates that it has the potential to provide substantial performance improvements. Users of migration in Sprite are enthusiastic about it, not just because it improves performance but also because evictions are so nonintrusive. Migration has not only been accepted in Sprite, it is now taken for granted.

In the future, the computing power of a single host is likely to continue to increase dramatically, but there will always be individual applications with the ability to use processors beyond those on a single host. The goal of future systems in this respect will be to help applications make full use of a system's resources in a fair and efficient manner. Process migration will be an important tool for meeting that goal.

Appendix A

System Call Management

When a process executes on a remote host, it performs system calls by trapping into the kernel on the remote host. As described in Section 4.3.2, in most cases the remote kernel services the call completely on its own, with no need to contact the process's home machine. In some cases, the remote kernel forwards the call to the home machine, and the call is processed completely by the kernel on the home machine. Normally, the call and its arguments are encapsulated a single procedure, with no interpretation performed by the remote host. Finally, in a few cases a kernel call requires action by both the remote host and the home machine. This action may be a complex sequence of RPC's (for example, the *wait* call obtains information about exiting child processes, and it may put the foreign process to sleep if no exiting children exist) or a simple update of a field in the process's process control block.

This appendix lists how each system call is handled in Sprite to ensure transparent process migration. Because Sprite attempts to be compatible with 4.3BSD UNIX, and UNIX is more widely known than Sprite, I list the system calls available in 4.3BSD UNIX [Com86]. All system calls that are available in Sprite but have no UNIX equivalent are handled transparently by the remote host, with one exception: the call to initiate migration is forwarded home, since it affects processes relative to their home machine.

Name	Disposition	Comments
accept	handle locally	uses file system connection to internet server
access	handle locally	talks directly to file servers
acct	unimplemented	
adjtime	unimplemented	
bind	handle locally	uses file system connection to internet server
brk	handle locally	state transferred at migration time
chdir	handle locally	talks directly to file servers
chmod	handle locally	talks directly to file servers
chown	handle locally	talks directly to file servers
chroot	unimplemented	
close	handle locally	talks directly to file servers
connect	handle locally	uses file system connection to internet server
creat	handle locally	talks directly to file servers
dup	handle locally	local bookkeeping
execve	handle jointly	updates name of program invoked
exit	handle jointly	clean up state on both hosts
fcntl	handle locally	talks directly to file servers
flock	handle locally	talks directly to file servers
fork *vfork*	handle jointly	initialize state of new process on home machine
fsync	handle locally	talks directly to file servers
getdtablesize	handle locally	location-independent
getgid	handle locally	state transferred at migration time
getgroups	handle locally	state transferred at migration time
gethostid	unimplemented	

Name	Disposition	Comments
gethostname	handle locally	remote host returns home machine and current execution site; library returns home machine by default
getitimer	handle locally	state transferred at migration time
getpagesize	handle locally	location-independent: processes migrate only between machines with the same page size
getpeername	handle locally	uses file system connection to internet server
getpgrp	blindly forward home	home machine maintains state of processes that may be distributed on different hosts
getpid	handle locally	location-independent
getpriority	handle locally	state transferred at migration time
getrlimit	unimplemented	
getrusage	handle jointly	remote host knows process's current usage, but home machine stores usage of children
getsockname	handle locally	uses file system connection to internet server
getsockopt	handle locally	uses file system connection to internet server
gettimeofday	blindly forward home	clocks are not synchronized
getuid	handle locally	state transferred at migration time
ioctl	handle locally	talks directly to file servers
kill	handle locally	location-independent
killpg	blindly forward home	home machine maintains state of processes that may be distributed on different hosts
link	handle locally	talks directly to file servers
listen	handle locally	uses file system connection to internet server
lseek	handle locally	talks directly to file servers (when offsets shared)
mkdir	handle locally	talks directly to file servers
mknod	blindly forward home	devices may be specific to host and must be created in the context of the home machine.
mount	blindly forward home	mounts disk on home machine
open	handle locally	talks directly to file servers
pipe	handle locally	state maintained by remote host

Name	Disposition	Comments
profil	handle locally	managed by current execution site
ptrace	unimplemented	
quota	unimplemented	
read	handle locally	talks directly to file servers
readlink	handle locally	talks directly to file servers
reboot	handle locally	intentionally nontransparent (calls for system administration not forwarded)
recv	handle locally	uses file system connection to internet server
rename	handle locally	talks directly to file servers
rmdir	handle locally	talks directly to file servers
select	handle locally	talks directly to file servers
send	handle locally	uses file system connection to internet server
setgroups	handle locally	state transferred at migration time
setpgrp	handle jointly	home machine updates copy of state
setquota	unimplemented	
setregid	handle locally	state transferred at migration time
setreuid	handle jointly	home machine updates copy of state
shutdown	handle locally	intentionally nontransparent
sigblock	handle locally	managed by current execution site
sigpause	handle locally	managed by current execution site
sigreturn	handle locally	managed by current execution site
sigsetmask	handle locally	managed by current execution site
sigstack	handle locally	managed by current execution site
sigvec	handle locally	managed by current execution site, transferred during migration
socket	handle locally	uses file system connection to internet server

Name	Disposition	Comments
socketpair	handle locally	uses file system connection to internet server
stat	handle locally	talks directly to file servers
swapon	unimplemented	
symlink	handle locally	talks directly to file servers
sync	handle locally	talks directly to file servers
syscall	unimplemented	
truncate	handle locally	talks directly to file servers
umask	handle locally	state transferred at migration time
unlink	handle locally	talks directly to file servers
utimes	handle locally	talks directly to file servers
vhangup	unimplemented	
wait	handle jointly	home machine maintains state about exiting processes
write	handle locally	talks directly to file servers

Appendix B

Compilation Speedup

This appendix contains a table with the data corresponding to Figure 7.3 on page 85.

Program	Number of Hosts 1	2	4	6	8	10	12
gremlin	1.00	1.63	2.67	3.40	4.03	4.21	4.43
kernel	1.00	1.69	2.75	3.59	3.96	4.13	4.35
pmake	1.00	1.55	2.29	2.75	2.98	2.91	2.95
TeX	1.00	1.81	3.09	3.91	4.67	5.18	5.42

Table B.1: *Speedup of compilations using a variable number of hosts.*

Bibliography

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Conference*, July 1986.

[AF89] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–56, September 1989.

[AK88] R. Alonso and K. Kyrimis. A process migration implementation for a unix system. In *Proceedings of the USENIX 1988 Winter Conference*, pages 365–372, Dallas, TX, February 1988.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conference*, Atlantic City, N.J., April 1967.

[Baa86] E. H. Baalbergen. Parallel and distributed compilations in loosely-coupled systems: A case study. In *Proceedings of Workshop on Large Grain Parallelism*, Providence, RI, October 1986.

[BBNG+89] A. Barak, R. Ben-Nattan, S. Guday, L. Picherski, O. Sasson, and E. Siegelmann. Running distributed applications under the MOSIX multi-computer system. Technical Report 89-15, Hebrew University of Jerusalem, November 1989.

[Ber85] Brian Bershad. Load balancing with *maitre d'*. Technical Report UCB/CSD 86/276, University of California, Berkeley, CA, December 1985.

[Bir85] K. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, December 1985. ACM.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BS85] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software—Practice and Experience*, 15(9):901–913, September 1985.

[BSW89] A. Barak, A. Shiloh, and R. Wheeler. Flood prevention in the MOSIX load-balancing scheme. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):23–27, Winter 1989.

[Cab86] L.-F. Cabrera. The influence of workload on load balancing strategies. Technical Report RJ5271, IBM Almaden Research Center, August 1986.

[Che87] A. R. Cherenson. The sprite internet protocol server. Master's thesis, Computer Science Division, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, 1987.

[Che88] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[Com86] Computer Science Division, University of California, Berkeley. *UNIX User's Reference Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, April 1986.

[Coo90] R. Cooper. Personal Communication, 1990.

[Dan82] R. Dannenberg. *Resource Sharing in a Network of Personal Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1982. Report No. CMU-CS-82-152.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.

[ELZ88] D. L. Eager, E. D. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *ACM SIGMETRICS 1988*, May 1988.

[Fel79] S. I. Feldman. Make — a program for maintaining computer programs. *Software—Practice and Experience*, 9(4):255–265, April 1979.

[FH89] C. J. Fleckenstein and D. Hemmendinger. Using a global name space for parallel execution of UNIX tools. *Communications of the ACM*, 32(9):1085–1090, September 1989.

[GC89] Cary G. Gray and David R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 202–210, December 1989.

[KL88] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 123–130, San Jose, CA, June 1988. IEEE.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lit87] M. Litzkow. Remote UNIX. In *Proceedings of the USENIX 1987 Summer Conference*, June 1987.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, June 1988. IEEE.

[ML87] M. Mutka and M. Livny. Profiling workstations' available capacity for remote execution. In *Performance '87, Proceedings of the 12th IFIP WG 7.3 Symposium on Computer Performance*, pages 529–544, Brussels, Belgium, December 1987.

[Mor] J. Morris. Confirmed via personal communication.

[MvRT+90] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[Nel88] M. N. Nelson. *Physical Memory Management in a Network Operating System.* PhD thesis, University of California, Berkeley, CA 94720, November 1988. Available as Technical Report UCB/CSD 88/471.

[Nic87] D. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12, Austin, TX, November 1987. ACM.

[Nic90] D. Nichols. *Multiprocessing in a Network of Workstations.* PhD thesis, Carnegie Mellon University, February 1990.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[PPTT90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *UKUUG Summer 1990 Conference Proceedings*, pages 1–9, London, England, July 1990.

[PW85] G. J. Popek and B. J. Walker, editors. *The LOCUS Distributed System Architecture.* Computer Systems Series. The MIT Press, 1985.

[RE87] E. Roberts and J. Ellis. *parmake* and *dp*: Experience with a distributed, parallel implementation of make. In *Proceedings from the Second Workshop on Large-Grained Parallelism*. Software Engineering Institute, Carnegie-Mellon University, November 1987. Report CMU/SEI-87-SR-5.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, pages 119–130, June 1985.

[SI89] J. M. Smith and J. Ioannidis. Implementing remote fork() with checkpoint/restart. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):15–19, Winter 1989.

[Sun87] Sun Microsystems. *Sun Release 4.0 Commands Reference Manual*, 1987.

[SvE89] A. Stolcke and T. von Eicken. Distributed probabilistic load information management in sprite. Term project, Computer Science 262, University of California, Berkeley, May 1989.

[The86] M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems*. PhD thesis, Stanford University, 1986.

[TL88] M. Theimer and K. Lantz. Finding idle machines in a workstation-based distributed system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 112–122, San Jose, CA, June 1988. IEEE.

[TLC85] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 2–12, December 1985.

[Wel86] B. B. Welch. The Sprite remote procedure call system. Technical Report UCB/CSD 86/302, Computer Science Division, EECS Department, University of California, Berkeley, June 1986.

[Wel90] B. B. Welch. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, CA 94720, February 1990. Available as Technical Report UCB/CSD 90/567.

[WM89] B. J. Walker and R. M. Mathews. Process migration in AIX's transparent computing facility (TCF). *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, 3(1):5–7, Winter 1989.

[WO88] B. B. Welch and J. K. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *USENIX 1988 Summer Conference*, pages 37–49, San Francisco, CA, June 1988.

[Zay87a] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–22, Austin, TX, November 1987.

[Zay87b] E. Zayas. *The Use of Copy-On-Reference in a Process Migration System.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, April 1987. Report No. CMU-CS-87-121.

[Zho87] S. Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems.* PhD thesis, University of California, Berkeley, CA, October 1987. Technical Report No. UCB/CSD 87/376.