Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems

by

Marvin M. Thcimcr

Department of Computer Science

Stanford University
Stanford, CA 94305

Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems

Marvin M. Theimer

Abstract

A loosely-coupled distributed system consisting of a cluster of workstations and server machines represents a large amount of computational power, much of which is frequently idle. Users would like to take advantage of this idle processing power by running one or more jobs in parallel on underutilized workstations. The use of underutilized workstations as computation servers not only increases the processing power available to users, but also improves the utilization of the hardware base.

However, this use must not compromise a workstation owner's claim to his machine: A user must be able to quickly reclaim his workstation to avoid interference with personal activities, implying removal of "guest" programs within a few seconds time. In addition, use of remote machines as computation servers should not require programs to be written with special provisions for executing remotely. That is, remote execution should be *preemptable* and *transparent* On the other hand, rather than simply terminate the guest program it should be possible to migrate it to another available workstation.

In this thesis, we study the key design and performance issues that affect preemptable remote execution in a loosely-coupled distributed system. Five major topics are addressed in our work: (1) provision of network-transparent execution environments for programs, (2) structuring migration facilities such that they interfere with the normal operation of the system in a minimal manner, (3) elimination of residual dependencies that occur when a program migrates but has state information left in machine-relative servers on the original machine, (4) provision of global scheduling facilities for finding idle/lightly loaded machines for remote execution and migration of programs, and (5) provision of fair access to global resources among the programs and users of a system. In the process of addressing these topics we delineate when remote execution facilities, with or without migration facilities, are useful and under what conditions they are easy (or difficult) to provide.

This research was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431. This technical report reproduces the author's Ph.D. thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A distributed system consisting of a cluster of workstations and server machines represents a large amount of computational power, much of which is frequently idle. For example, our research system consists of about 40 workstations and server machines, providing a total of about 50 MIPS. With a personal workstation per project member, we observe that over one third of our workstations are idle, even at the busiest times of the day.

There are many circumstances in which the user can take advantage of this idle processing power. For example, a user may wish to compile a program and reformat the documentation after fixing a program error, while continuing to read mail. In general, a user may wish to run batch jobs concurrently with some unrelated interactive activity. Although any one of these programs may perform satisfactorily in isolation on a workstation, forcing them to share a single workstation degrades interactive response and increases the running time of non-interactive programs.

Users may also have distributed applications they wish to run in order to gain even more performance benefits. For example, parallel recompilation of the individual modules of a program can sometimes provide an order of magnitude speed-up if sufficient idle workstations are available. Even if dedicated computation servers are available, they may not be able to provide the processing power available to some applications through parallel execution.

Use of idle workstations as computation servers increases the processing power available to users and improves the utilization of the hardware base. However, this use must not compromise a workstation owner's claim to his machine: A user must be able to quickly reclaim his workstation to avoid interference with personal activities, implying removal of remotely executed programs within a few seconds time. By removal, we mean that the remotely executed programs will be migrated to other workstations.[1] In addition, use of remote machines as computation servers should not require programs to be written with special provisions for executing remotely. In short, remote execution should be *preemptable* and *transparent*

In this thesis, we examine the issues governing preemptable remote execution facilities and present design recommendations for distributed operating systems that would support such facilities. We argue that remote execution allows idle workstations to be used as a "pool of processors" without interfering with use by their owners and without significant overhead for the normal execution of programs. In essence, we seek to extend one of the principal benefits of traditional time-sharing systems into a distributed environment, namely resource sharing at the processor and memory level. We conclude that the cost of providing preemption is modest compared to providing a similar amount of computation service by dedicated computation servers.

[1] Both the "owner(s)" of a program and the "owner(s)" of the workstation it is running on should be able to demand migration of the program.

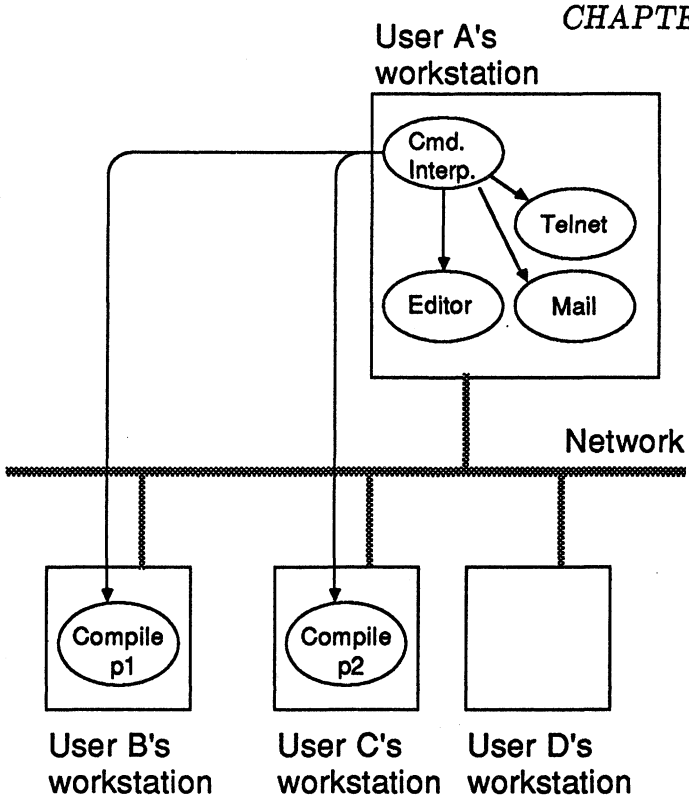

Figure 1.1: Remote execution of User A's compilations.

1.1 Example Scenario

As an example of how preemptable remote execution would work, consider the scenario depicted in Figure 1.1: User A has just finished modifying a program and wishes to recompile the program and update its documentation in a database of such information. He runs his documentation update program locally to modify the program's documentation and remotely executes the program recompilation in order to avoid resource contention between the two. The compilation involves recompiling two separate program modules, which results in parallel invocation of compile programs on two separate machines. The system's global scheduling facilities are employed to select two lightly loaded machines for these tasks; in this case, the machines of users B and C are picked. Machine B has no local user present and is completely idle at the time it is selected to be a remote execution host. Machine C *does* have a local user present who is editing a file, but is consuming only a small percentage of the machine's resources in doing so (say 3% of the CPU).

If the local operating system of each machine implements some form of priority-based round-robin scheduling, then the remotely executed compilation will not significantly affect the performance of User B's editor. But suppose that at some point during the compilation User B requires most of the machine's resources because his editor must reformat a document being edited. The "guest" compilation program is preempted to prevent contention for resources from starving it. Subsequently, the compilation program is migrated to another lightly loaded machine that is picked by the system's global scheduling facilities, in this case, user D's machine. Figure 1.2 depicts the resulting configuration.

Some points to note are that no user is aware of migration and that no user need perform any explicit actions to ensure privileged access to their own machine's resources. However, nothing prevents a user or program from explicitly invoking migration. Thus, a user could manually preempt all guest programs from his machine even when his locally

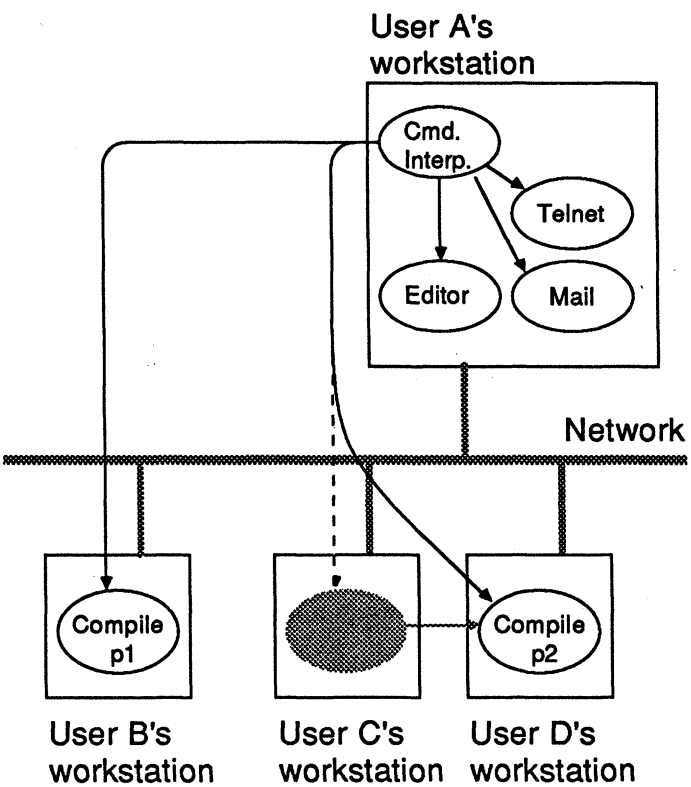

Figure 1.2: Preemptive migration of User A's compilation.

generated load would not cause the operating system to do so automatically. He could also migrate his own locally executing programs to remote machines; for example, after they had been preempted back to his own machine because no other idle machines were available earlier. Similarly, a program could ask the operating system to migrate it elsewhere if it so desired.

1.2 Computational Model

1.2.1 Program State

The model of computation we will use treats programs as one or more address spaces, each of which may contain one or more processes—meaning threads of execution. Thus, a program's local state consists of the state of the code and data in these address spaces and the state of each process in them. We assume that the local operating system on each machine provides multiple virtual address spaces, parts of which may be stored in a machine's main memory and parts of which may reside on secondary "backing" storage. Typically, the states of processes are kept in the kernel of the local operating system of the machine where a program is running.

We will also assume a *client/server* model of interaction in which all services in the system are provided by server processes that manage resources on the behalf of clients. Clients and servers are assumed to exist in their own, separate address spaces so that remote execution and migration of programs does not require provision of cross-machine shared memory semantics. Systems in which multiple programs share common data structures (such as the shared file descriptors provided by UNIX[2]) will not be considered.

One implication of the assumption of separate address spaces is that communication between programs must be done by some form of *message-passing*, where message-passing is defined for the moment to be just copying of a data segment from a specified location in a sender's address space to a specified location in a receiver's address space. This "basic" definition of message-passing includes such communication paradigms as UNIX *pipes*, for example.[3] We will place no restrictions on the form of message-passing provided by a system other than to require that the operating system perform the actual copying of the data; which, in turn, implies that communications must take the form of a sender process invoking an appropriate "system call" to initiate a message-passing operation and a receiver process invoking an equivalent "system call" to receive the message data. This restriction will be necessary for implementing communications between programs that can migrate. It implies that we forbid programs that communicate, for example, by means of reliable byte streams implemented directly on top of the network interface of a machine instead of on top of the message-passing facilities of the machine's operating system. However, any communications protocols implemented on top of the system's message-passing facilities are allowed, including "datagrams", asynchronous reliable messages, synchronous messages (the sender blocks until the receiver sends a reply message), reliable byte streams, pipes (buffered reliable byte streams), remote procedure calls, etc. We also deal with, and make good use of, *broadcast*[MB76,Wal80] and *multicast* (broadcast to a designated subset of all possible receivers) messages, with no restrictions placed on whether the set of receivers is known to the sending process or to the operating system on its machine.

Although clients and servers may not share memory segments, we do allow servers

[2]UNIX is a trademark of AT&T Bell Laboratories.

[3]In principle, one can also implement such things as shared UNIX file descriptors with message-passing by making the file descriptors locked objects that are "passed" back and forth between address spaces as needed. The LOCUS[PWC*81] operating system (see Chapter 2) provides this facility. However, it is unclear how this would affect the performance of various applications and we will not consider such forms of communication in our model.

to maintain state for clients, so that the state of a program consists of its "local" state described above and the state kept for it in various servers throughout the distributed system. For example, each machine might have a name server that stores name bindings specific to the programs running throughout the system on behalf of the machine's local user. Each machine might also have a memory manager that implements the virtual memory support for all locally running programs. Finally, network file servers might store files (including temporary files) for programs running throughout the system.

We also make a distinction between *machine-relative* servers, which provide services that are local to, and hence, are replicated on, each machine, and *global* servers, which provide services that are, at least logically, centralized and are visible to clients throughout the distributed system. (We do not preclude machine-relative servers from being globally visible.) When a program is migrated from one machine to another the state information kept for it in machine-relative servers will have to be modified (typically by migrating it as well) to reflect the changed location of the program. In contrast, state kept for a migrating program in global servers will not have to change.

1.2.2 Failure Model

Fault tolerance and recovery of programs from machine failures are regarded as application-level issues that will not be covered in this thesis. Thus, neither issues such as whether or not to remotely execute a program or subprogram on the basis of fault tolerance considerations nor issues such as failure recovery of the subprograms of a distributed application will be addressed. Likewise, Byzantine failure considerations[LSP82] will also not be covered.

In general, we will assume that most applications are idempotent in nature, that is, that they can simply be rerun if they fail, so that failure recovery facilities are *not* necessary for normal system usage. If fault tolerance were a requirement for most of the applications in a system, then migration of programs between machines could be implemented by simply destroying a program on one machine and recovering it on another machine. Our goal is to provide the users of a system access to additional resources rather than greater failure resiliency and our designs will emphasize low cost over provision of fault-tolerance.

An area where we *must* deal with fault tolerance is that of reliable communications. Communications facilities can be provided at two levels. The operating system provides communications facilities used by all applications programs in the system. On top of these, applications themselves can provide additional communications protocols. If the operating system guarantees reliable delivery of messages, then it must do so even if remote execution and migration of programs is allowed. On the other hand, if applications provide additional communication protocols outside the "knowledge" of the operating system, then they are responsible for the fault tolerance of those communication protocols. This implies that whenever the system's underlying communications behavior changes, it may invalidate the assumptions used by various applications for their own communications protocols.

Our model of reliable operating system-level message communications (if reliable message passing is provided) is that the operating system will attempt to retransmit a message to its (remote) receiver n times, with retransmissions occurring if the receiver's machine's operating system has not sent an acknowledgement message of some sort within a time t_r after the previous attempt to (re)send the message. If no acknowledgement/response message is received after n retransmissions, the receiving process is assumed to be dead and the sender of the message is notified of the failure. We will treat network partitioning as an application-level issue that is not handled by the operating system.

The retransmission interval t_r is allowed to be any function of the number of retransmissions sent so far and the id/location of the sending and receiving parties, subject only to the requirement that the time required for a sender to receive notification that a receiver is dead is "short". Specifically, message operations to nonexistent processes are assumed

to terminate within a time that is at most a few seconds. We assume that retransmission stops after the sender has been notified of failure. Consequently, applications may assume that communications operations will terminate within a bounded, relatively short amount of time.

Application-level communication protocols must also deal with the possibility of timeouts, both at the level of timeouts returned from the underlying interprocess communication (IPC) facilities and at their own level of interaction. We place no restrictions on application-level protocols in our model of computation, implying that applications may incur two types of "incorrect" communication timeouts:

- In point-to-point connections, a response to a sender may timeout in the sense that the receiver of a message may simply not reply in time, even though the receiver is alive as far as the operating system is concerned (and the sender could query the operating system to find this out).
- If broadcast/multicast communication is used to communicate with an unknown set of receivers, then the sender of a message *must* use a timeout value to decide when no more replies are to be expected since the operating system may not be able to determine the state of all receivers. If a replier delays too long before replying, the sender will incorrectly decide that no more replies are forthcoming.

1.2.3 Network Transparency and Remote Execution

There are many possible definitions for "network transparency" and "transparent remote execution". For example, we might define network transparency as a condition in which programs *cannot* determine their location. Unfortunately, this is not always possible in the sense that some programs intentionally generate location-dependent output or incorporate location-dependent behavior. For example, a program that prints out the network address of the machine it is running on cannot be written for a system that enforces this definition of transparency.

Our goal for remote execution is not this kind of network transparency; we wish to allow the existence of programs whose output or behavior is location-dependent *if that is their intended effect.* We would like to require that programs that were originally intended to be location-independent continue to be location-independent in an environment that supports remote execution. Unfortunately, defining what it means for a program to be "location-independent" is not an easy matter. If one attempts to define it in terms of the "output" of the program then there are difficulties with defining exactly what the output of a program is. For nonterminating programs the definition clearly should include considerations about their intermediate states. But this raises questions of how the intermediate states of programs should be treated in general and how the state of "unimportant" data, such as the values in data structures returned to a global free list, should be taken into account.

We will circumvent these problems by requiring a fairly restrictive form of location independence that is based on the sequence of operations a program executes.[4] Specifically, we define a class of programs that obey the following conditions on their initial state and the operations they execute:

1. The program starts in a state that is independent of its location.
2. The program executes only operations whose effect on the state of the system, i.e.

[4]Depending on the setting, the term "operation" can be taken to mean anything from the microcode instructions available on some piece of hardware to the set of commands available from a command interpreter. We shall take the term to mean the set of instructions executed by a program exterior to the library routine interface to the operating system. Thus, an operation might be a single machine instruction, such as an add instruction, or it might be a subroutine call to a library routine that perhaps invokes a system call into the kernel of the operating system.

whose output, is independent of the speed of the processes that are executing the operations. We assume that operations have no visible intermediate state.

3. The program executes only operations whose output is independent of the location at which they are executed.
4. The program executes only operations whose output is independent of the state of the system exterior to the program's internal state.

Such programs execute a sequence of operations whose order is independent of the location at which the program is run. We shall say that a system whose remote execution facilities allow a program in this class to execute correctly on any host, independent of whether the program is executed locally or remotely with respect to its invoker, provides *transparent remote execution.*

To see why we must so tightly restrict the class of programs for which location independence is ensured, consider the following program examples, all of which are location dependent in nature. The first example, corresponding to the second condition above, is a program that prints the time of a clock at a particular host in the system. This program will print a different sequence of values, depending on the time needed by the system's communications facilities to perform the query operation of the clock. Unless both local and remote query operations take the same amount of time to execute, the location of the program with respect to the host with the clock will influence the performance of the query operations and, hence, the output of the program. In general, we cannot assume that local and remote operations will perform at the same speed. The second example, corresponding to the third condition above, is a program that prints the network address of the host it is executing on. Depending on the location it is executed at, a different value will be printed. The third example, corresponding to condition four above, is a program that first destroys all other programs in the system and then prints the load on a particular host. Again, different values will be printed, depending on its location with respect to the host whose load is being printed out.

To provide transparent remote execution, we need to ensure that the set of operations that are independent of processing speed, location, and state of the system external to the program executing them are *network-transparent.* That is, they can be executed correctly independent of the location at which they are executed and the location of their operands.

We define the term *execution environment* to mean the names, operations, and objects with which a program wishes to interact during execution. We define the term *network-transparent execution environment* to mean an execution environment in which all names, operations, and objects are accessed in the same manner irrespective of their location with respect to the program accessing them. Furthermore, we require that all operations whose output is not a function of the state of the system's communications facilities to produce the same output irrespective of whether they are invoked locally or remotely with respect to the objects operated on. Provision of a network-transparent execution environment guarantees the provision of transparent remote execution since *every* operation is network-transparent. Note, however, that the converse is not true; provision of transparent remote execution does not guarantee a network-transparent execution environment. This is because transparent remote execution places no constraints on the transparency of operations whose output is location dependent. Thus, for example, the operation of printing the local host's network address need not be network-transparent because programs that employ this operation do not belong to the class of programs for which transparent remote execution is guaranteed. In this thesis, we will restrict ourselves to discussing the provision of network transparency for *all* operations provided by a system, realizing that this requirement need not be met for all operations if just transparent remote execution is to be achieved.

Finally, there are two additional goals we wish to meet with our remote execution facilities. First, although our definition of transparent remote execution does not place any constraints on performance of local versus remote execution, we would like to limit the performance difference between local and remote execution where possible. Consequently,

part of this thesis will concern itself with some of the issues relating to performance of remotely executed operations. Second, while we ensure *transparent* remote execution only for the class of programs define above, we would like to permit *any* program to be executed remotely. Thus, for example, the user is free to take advantage of location-specific information if he so desires.

1.2.4 Workload Characterization

We are primarily interested in a workstation-based, environment. Such an environment is not principally aimed at supporting compute-intensive, long-running applications. Instead, the workload consists primarily of interactive and "fast-turnaround" applications that exist for relatively short periods of time.[5] Furthermore, the total number of such applications running in the system varies considerably, depending on how many users are present and active at any given time. The result is a workload that usually leaves many machines "idle", where idle is defined here to mean a level at which low-cost activities such as text editing—which is the dominant activity of most users—are still handled. However, the workload can still vary greatly in magnitude over time and is *not* homogeneous in nature. Groups of users may go to lunch or a group meeting; at night, a few users may grab the entire system's free resources to run large, distributed computations. This contrasts with an environment in which compute-intensive, long-running applications take up all available resources in the system, thereby keeping the workload uniformly high.

1.3 Additional Assumptions

In order to migrate preempted programs between machines in the middle of execution, we will assume an environment consisting of homogeneous hardware so that the representation of a program's state (external to the operating system at the library routine interface, as described in Section 1.2.3) will be the same on both the machine it is originally executing on and the machine it is migrated to. If several kinds of machines exist within a system, then migration can be performed among any two machines of the same kind. We will not consider the question of migration between dissimilar machines in this thesis.

We also assume that the communications medium provided between machines of the system has the bandwidth and latency characteristics of contemporary local area networks, so that remote communications—most importantly remote program loads—are not prohibitively expensive to perform. Specifically, we assume a communication medium that can provide multi-megabit communication channels with message latencies on the order of milliseconds and tens of milliseconds rather than seconds.

Several parts of this thesis will rely extensively on the availability of efficient multicast communications facilities. Although preemptable remote execution can still be provided without multicast facilities, failure considerations force rebinding references to migrated programs to become a considerably more difficult problem. Since most local area networks provide efficient broadcast/multicast facilities in hardware, we consider the availability of efficient multicast facilities built on top of these to be a reasonable assumption to make. We will only briefly discuss how to implement preemptable remote execution facilities in an environment without multicast.

A necessary corollary of our assumptions about communications media is that our work is applicable principally to distributed systems that span a single local network or perhaps a local internet consisting of a few such networks connected by high-speed gateway machines. In line with our assumptions about the availability of efficient multicast communications, we will *not* consider system partitioning issues, so that (possibly retransmitted) multicast

[5] **Examples include text editing, electronic publishing, program development, and CAD applications.**

messages can always be assumed to eventually reach every functioning machine in the system. Issues of extension to an internet environment will be partially addressed in the future work chapter of the thesis.

Finally, we will assume the availability of a distributed operating system that provides concurrently executing processes that communicate in a location-transparent manner via messages. The message-based communications need not be visible to applications, perhaps being hidden under the veneer of a remote procedure call mechanism or reliable byte streams, for example. We will not discuss how such an operating system can be provided except to give references to existing operating systems in Chapter 2.

1.4 Key Issues

The issues in the provision of preemptable remote execution facilities can be divided into three broad areas:

- Remote execution on a particular machine.
- Migration of a program from one machine to another.
- Selection of a suitable machine for remote execution or migration.

1.4.1 Remote Execution

As discussed in Section 1.2.3, we wish to provide transparent remote execution by providing programs with a network-transparent execution environment. In this section we will describe several requirements that must be met in order to provide a network-transparent execution environment.

Programs reference services/objects that exist in both local and global *contexts:* Some services/objects are provided by servers whose "scope" is local to the machines on which they run whereas others may be provided by servers that are global in scope. An example of the former is a resource manager that controls the resource allocations for programs running on its machine. An example of the latter kind of server is a network file server. Care must be taken to properly delineate these two types of references in order to obtain proper binding of references to their intended objects. The difficulty lies in the fact that, at some level, local services must be referred to by means of *machine-relative* names. Stated differently, we must bind a particular name to a particular instance of a service that exists on each machine. Furthermore, a machine-relative reference should sometimes be interpreted relative to another machine, for example, the machine at which the human user who ultimately invoked the program is sitting. In general, one needs to define several "naming contexts" which allow programs to refer to things relative to a particular logical entity without explicitly knowing its actual physical location. The *closure* of such naming contexts represents the correct set of bindings to use for the name space of a program's execution environment[Sal79].

Another kind of problem that must be dealt with is direct access of programs to hardware devices, such as a graphics frame buffer, which may be inefficient, or impossible in practice, to provide remotely.

1.4.2 Migration

The principal concern with migration of a program is the interference it causes with the execution of the program and with the rest of the system. This interference takes on two major forms:

- Migration of a program requires atomic transfer of a copy of the program's state to another machine.
- The migrated program may continue to depend on its previous host once it is executing on a new host; that is, it may have *residual dependencies* on the previous host.

Atomic transfer of program state is required so that the rest of the system at no time detects the existence of more than one copy. However, suspending the execution of the migrating program or the interactions with the program for the entire time required for migration may cause interference with system execution for several seconds and may even result in failure of the program. We consider this unacceptable, which implies that such long "freeze times" must be avoided.

Residual dependencies occur because of state stored outside a program's address spaces on the previous host. Some of this state—for example, exception handler bindings kept in a machine-relative exception server—*must* be migrated in order for the program to continue executing properly.

Other state, such as temporary files created within a local file server, need not be migrated to maintain correctness. However, if these files are not migrated along with the program, the migrated program continues to interfere with its previous host, contrary to the purpose of migrating the program. Also, a failure or reboot of the previous host would cause the program to fail because of these inter-host dependencies.

1.4.3 Global Resource Scheduling

Finding a suitable host to execute on, or to migrate to, is a question of global resource scheduling. Facilities that support this should provide:

- Available and efficient selection of hosts for remote execution and migration.
- Some level of fair access and contention resolution for resources among programs and users.

The minimum scheduling facility required to support remote execution is the ability to select a remote host and contact it (to request or negotiate for execution privileges). Since remote execution cannot proceed without this facility it should be made highly available. Efficiency is obviously desirable, although if the total overhead incurred is small, then efficiency may not be an issue. If hosts are not 100% utilized, they may be able to absorb the overhead into their idle time.

Although not strictly necessary, the ability to monitor and enforce access policies to resources is very desirable.[6] Within a single machine, this can be enforced by techniques such as round-robin scheduling, giving locally originating programs priority over guest programs, and by preempting guest programs in order to force them off the machine. On a more global scale, it is desirable to achieve some level of fairness among users' access to the resources of the system as a whole. If idle machines are scarce, as may happen if large distributed applications are run, then we would also like to gain quick access to them when they become available.

Three major issues must be addressed in designing scheduling facilities:

- differing workload assumptions,
- communications model employed,
- scalability.

[6] A well-known system that does *not* impose many fairness constraints is UNIX. By running jobs concurrently, a user can grab a large fraction of a host's CPU cycles.

In Section 1.2, we outlined a workload consisting of interactive and "fast-turnaround" applications that are run in a very inhomogeneous environment. Depending on how heavy the workload is, how compute-bound most applications are assumed to be, and how many distributed applications of various kinds are run, very different scheduling facilities will be appropriate. For example, if most machines are under-utilized most of the time, then very simple scheduling facilities will be sufficient and fairness will be a non-issue. Conversely, if we assume that the system load will be high—for example, because some users begin running large distributed applications on a regular basis—then the global scheduling facilities will have a harder time finding idle resources and will also have to take fairness considerations into account. If the distributed applications run are sensitive to variations in the resource shares received by each component, then the system should worry about fairness at the "microscopic" level in addition to achieving fairness among users over the long run.

In an environment in which the set of machines available as "remote execution servers" is constantly changing, a communication model based on some form of broadcast can provide a much more efficient means of disseminating information or of collecting information than can be provided by point-to-point communications. However, indiscriminate use of broadcast can inflict a significant overhead on the system, particularly on hosts that are not interested in participating in the activity using broadcast (e.g., global scheduling). The provision of multicast communication facilities can alleviate much of the overhead problem by directing communications only to interested parties in the system.

Both broadcast and multicast bring with them their own set of issues to consider. Reliable broadcast/multicast can be expensive to implement. If unreliable broadcast/multicast is used, then algorithms must take this into account. Furthermore, the loss rate of replies to query messages can be very significant if the number of replies generated to a query is larger than the buffering capacity of the sender's hardware and/or software. With multicast, one must also worry about how to manage the membership of machines in multicast groups.

Distributed systems can vary enormously in size. A small system may contain only two hosts (e.g., if all others have failed), whereas a large system might consist of several hundred to a thousand hosts. Size affects not only potential bottlenecks such as central servers, it also affects reliability and overhead issues for multicast communications and decentralized facilities that wish to maintain a distributed information base.

In many ways, scalability is the only real issue since for small systems one can use "brute-force" techniques to deal with most other issues. For example, in small systems, one can frequently afford to maintain nearly complete global state information at each host, even using point-to-point communication, because the overhead and response delays incurred are small enough to be acceptable.

1.5 Topics Covered

This thesis is primarily concerned with issues of mechanism and not policy. Our goal is to study remote execution and preemption facilities that can support various policies, not to study the policies themselves. Considerable research has already been done in the area of scheduling policies for distributed systems. We will make policy assumptions only as they are necessary for delineating design considerations for the facilities we wish to provide. The principal issues we will examine concern the mechanics of actually providing facilities for remote execution and migration of programs onto specific hosts and managing the size of the scheduling information that must be dealt with by global scheduling facilities.

Chapter 2 describes related work, both in the provision of facilities for remote execution and migration and in the area of scheduling policies for distributed systems.

Chapter 3 covers issues that must be dealt with when just remote execution is to be

provided in a system. Since remote execution by itself can provide considerable benefits and migration facilities may be difficult to provide in some systems, this is an important design possibility to consider. Chapter 4 will deal with the issues specific to the topic of preemptive migration.

Chapter 5 examines the question of how to locate idle host candidates for remote execution and migration. A principal aspect of our work in this area will be the assumption of efficient multicast communication facilities that allow, among other things, for the provision of both centralized and decentralized global scheduling schemes. A comparison of how both can make use of multicast and of their relative performance under various workload and scaling assumptions is the main focus of this chapter.

Chapter 6 addresses issues of fairness and load balancing. In line with our focus on mechanism rather than policy, this chapter emphasizes minimal mechanisms required to achieve various approximations of equal access to system resources by programs and users.

Chapter 7 describes an implementation of preemptable remote execution facilities that has been done for the V-System[7]—a message-based, distributed operating system designed primarily for high-performance workstations connected by local networks[Che84]. An overview of the implementation and a summary of both performance measurements and usage observations are presented in this chapter.

The conclusions drawn in this thesis are summarized in chapter 8. Chapter 8 also describes future work.

1.6 Topics Not Covered

When the programs of multiple users must coexist on the same machine, security becomes an important issue to consider. Unwanted interactions between a host and any remotely executing guest programs it has accepted must be prevented. One must worry both about corruption of the host by the guest programs and also about corruption of the guest programs by the host. This topic has already been investigated as part of another thesis[Dan82] and we will make use of the published results without modification. Chapter 2 gives a brief overview of these results.

We will also not deal with any issues of fault tolerance or failure recovery except in the discussion of future work. The discussion will center mostly on the suitability of migration facilities as building blocks for failure recovery rather than on how failures and failure recovery facilities affect remote execution and migration.

1.7 Research Contributions

In this thesis, we study the key design and performance issues that affect preemptable remote execution of programs in a loosely-coupled distributed system. We also provide design recommendations and requirements for building distributed operating systems that can support preemptable remote execution. In particular, five major topics are addressed in our work:

- ***Provision of network-transparent execution environments for programs.*** **We specify** requirements on operating system design that must be met in order to achieve transparent remote execution of the sort defined in Section 1.2.3.
- ***Structuring migration facilities such that they interfere with the normal operation*** *of the system in a minimal manner.* We show that migration facilities can be constructed such that they cause interference that is similar in scope to that caused by

[7] **V-System is a trademark of Leland Stanford Junior University.**

other operating system "events" such as temporarily swapping a program out to its backing store in a paging virtual memory system. We also specify the factors that determine the time required to migrate a program from a machine.

- *Elimination of residual dependencies that occur when a program migrates but has state information left in machine-relative servers on the original machine.* We specify requirements on server structure that must be met in order to migrate server state belonging to a migrated program. We also demonstrate that a system can be designed and built that avoids the major problems of residual dependencies and machine-relative server state by pushing the system's naming and file service facilities into the address spaces of a program and into global servers.
- *Provision of global scheduling facilities for finding idle/lightly loaded machines for remote execution and migration of programs.* The principal emphasis of this work is an examination and comparison of how centralized and decentralized scheduling facilities perform in an environment in which multicast communications are available. We show that, under the workload and environment assumptions we make, both centralized and decentralized scheduling facilities can be built that are scalable to systems containing hundreds of machines and that are highly available. We show that, under our assumptions, the performance and availability of both designs is roughly equivalent, with a centralized design providing additional scalability to several thousand machines in exchange for greater complexity of implementation. We conclude that the greater scalability is probably not worth the added complexity.
- *Provision of fair access to global resources among the programs and users of a system.* Arguments are presented for why this issue can be ignored in many cases. We show that preemptive migration by itself provides statistical fairness among users without starvation of any given program, assuming that all users generate an equal workload when averaged over a longer period of time. Additionally, a global scheduling policy is presented that bounds the disparity in CPU share received by any remotely executing program in the system to at worst slightly less than half the share received by the average remotely executing program in the system. We also show that a very simple load balancing scheme can be used to bound the disparities in CPU shares received by the independent subprograms of a distributed application even further in exchange for a potential loss in throughput of the system as a whole.

In the process of addressing these topics we delineate when remote execution facilities are useful and under what conditions they are easy or difficult to provide. Similarly, we describe when preemptive migration facilities are useful, when they can be omitted from a system, and for what kinds of systems they are easy or difficult to provide.

Finally an implementation of preemptable remote execution facilities, including both centralized and decentralized global scheduling facilities, has been completed to demonstrate its feasibility and to provide a quantitative characterization of the performance of such facilities.

Chapter 2

Related Work

The related work that is relevant to this thesis can be roughly divided into two categories:

- Efforts to extend distributed operating systems to provide remote execution and migration of programs.
- Investigation of algorithms for global load sharing.

A third category one might consider is (possibly distributed) applications that would run on top of our facilities. However, since we seek to be as application-independent as possible, this category is mentioned only in passing.

2.1 Distributed Operating Systems

The existence of several distributed operating systems that are in production use demonstrates the viability of the basic model of processes running in their own address spaces on arbitrary machines in a loosely coupled system. Several of these systems have provided remote execution and migration facilities, although they have made simplifying assumptions about their environment.

The principal result one can summarize from these existing (and some proposed) systems is that the basic mechanisms for remote execution and migration require that programs communicate with the rest of the system only through well-defined, and hence controllable, interactions such as messages. Difficulties arise whenever this paradigm is violated; for example, when shared memory semantics are involved or when communication information can be stored in arbitrary parts of a program's state.

Although several designs and systems have recognized the desirability of network-transparency, none have dealt with its implications for machine-relative servers and none have dealt with two of the key issues introduced in Chapter 1:

- All ignore the issue of unacceptable interference with the system as a whole due to suspension of activities during migration.
- All ignore the problem of residual dependencies on machine-relative servers (other than the operating system kernel) that occur as a result of migration of a program. Some systems, such as LOCUS[PWC*81,WPE*83,BP84], implement the entire operating system as a monolithic kernel with no machine-relative servers. However, these systems also have no concept of individual host autonomy, so that services such as the file system are treated as global in scope. As a consequence, programs see the same view of them no matter where they are running.

An area where different systems have opted for differing solutions is that of rebinding

references to migrated processes. Most solutions have either been failure-intolerant or degrade performance of the system even when migration is not involved.

The following subsections provide an overview of the various distributed operating systems that have provided remote execution and migration or are planning to. The design of most interest is Demos/MP, which has gone farthest in actually providing both remote execution and migration of programs. The design we present in this thesis is similar to that of Demos/MP in many ways. The overview of the other systems will concentrate on the differences from our work.

2.1.1 Demos/MP

Demos/MP[PM83] is a version of the Demos[BHM77,Pow77] operating system whose semantics have been extended to operate in a distributed environment. It is a message-based operating system in which a kernel on each host machine implements processes, messages, and message paths, called *links*. Most of the system functions are implemented in server processes, which are accessed through messages.

In Demos/MP, messages are sent using links to specify the receiver of a message. Links are protected global process addresses that are accessed via a local name space. Links may be created, duplicated, passed to other processes, or destroyed. They are manipulated much like capabilities; that is, the kernel participates in all link operations, but the conceptual control of a link is vested in the process that the link addresses (which is always the process that created it).

The provision of process migration[1] in this operating system required adding two additional features to message links. The first was an additional field for the link address, which contains a best guess of the location of the process. The second was the addition of a link attribute, called DELIVER-TO-KERNEL, which causes a link to refer to the kernel of the processor on which a particular process resides. This allows the system to address control functions to the kernel of a process without needing to know which processor the process is on (or moving to). For example, access to data segments specified in a link are achieved by means of kernel MOVEDATA calls which use this attribute to achieve location transparency.

The version of Demos/MP for which process migration was implemented was designed to run on a multi-processor machine rather than in a loosely-coupled distributed system of multiple machines. As a consequence, it does not have any machine-relative server processes other than the "pseudo" process representing the kernel. It also assumes that sufficient memory resources (e.g., through demand-paged virtual memory) are available to allow separate buffering of all messages sent to a process and of all data to be placed into its address space by MOVEDATA operations. These assumptions make the actual mechanics of process migration fairly straight-forward:

1. The process to migrate is suspended. The process ceases executing and messages and MOVEDATA operations are queued.
2. The destination kernel creates descriptors for an empty process state and then copies the state of the migrating process's descriptors on the source kernel into these using the normal MOVEDATA operation.
3. The destination kernel creates an empty address space and copies the contents of the address space of the migrating process to it.
4. Any pending messages and MOVEDATA operations are forwarded to the new processor.
5. The process's state on the source processor is destroyed and the new copy on the

[1] In Demos/MP, each process resides in its own address space.

destination processor is set running again.

In order that messages subsequently sent to the migrated process will find its new location, a *forwarding address* is left on the source processor. This is used to forward messages sent to the process's old location and also results in a "rebind" message to the sender's kernel telling it to reset the location field of its links for the migrated process.

While demonstrating that process migration facilities are reasonable to provide, the design makes several simplifying assumptions that are unacceptable in many circumstances. Summarizing the effects of these:

- The assumption of no machine-relative servers means that rebinding of *location-dependent* references did not have to be considered except for the special case of kernel references. Also, only the kernel's state for the migrating process had to be transferred. In a loosely coupled system, each machine might have, for example, a program manager and a memory manager which administer local execution of programs and use of memory, respectively.
- Since Demos/MP is targeted for a tightly-coupled hardware environment, there is no concept of autonomy and, therefore, no concept of residual dependencies on such things as file servers.
- An example of how migration can interfere with the system as a whole is illustrated by the decision to buffer messages and MOVEDATA operations for a suspended process. This decision implies that each processor must be able to buffer arbitrarily large amounts of data *outside* the process's address space. While this can be done if demand-paged virtual memory is available, it still implies "double-copy" semantics since the buffered data must be forwarded after migration has completed. This, in turn, increases the delays seen by the system in the responsiveness of the migrated process.
- Redirection of existing links employs a lazy-evaluation scheme that relies on the availability of the source processor to provide a forwarding address. Failure recovery facilities are assumed, which are able to restart crashed processors from their point of failure ([PP83]). However, this assumption implies the availability of facilities that are not easy to provide and that most systems do not have. Alternatives of returning messages as undeliverable or of querying a central repository of forwarding addresses are mentioned in the published literature, but were considered unacceptable by Powell et. al due to lack of transparency and poor performance.

2.1.2 LOCUS

LOCUS is a UNIX-compatible distributed operating system that allows the migration of both files and programs[PWC*81,WPE*83,BP84]. As in UNIX, the operating system is implemented almost entirely inside the distributed kernel, implying that the only machine-relative server on each machine is its local component of the kernel. Since LOCUS, like UNIX, is a multi-user time-sharing system, questions of autonomy and priority access to local resources is not a concern. Hence, residual dependencies to such things as file system components on the host a program was migrated from are considered acceptable.

As is the case for all other systems, no consideration is made for dealing with unacceptable latencies due to migration.

Being UNIX-specific also causes LOCUS some problems. Much discussion in the literature deals with the implicit shared-memory semantics that many UNIX inter-process functions have, such as shared file descriptors and pipes. A token mechanism is cited to resolve this issue: Only one process at any given time may access a shared resource, with a token being passed back and forth to control who currently has access privilege. It is claimed that access to shared resources, such as pipes, is typically done at a relatively course

granularity, so that this scheme works satisfactorily in practice. However, many systems simply disallow shared-memory items such as shared file descriptors.

2.1.3 Accent/Spice

Accent is a message-based operating system kernel designed at CMU to support the Spice project[RR81,FR85]. It is built around a single abstraction of communication between both processes and kernel functions such as device access and virtual memory management. Accent's design supports remote execution and process migration by providing controlled communication and encapsulation of processes (i.e., programs) in separate address spaces. This is achieved by forcing all interactions between processes (and their associated address spaces) to be performed by means of messages to *ports*, which can only be referenced by means of kernel-protected *capabilities* (similar to Demos/MP links). Messages are structured, typed, and variable-length in size. Access to the data in another process's address space is obtained by having that process send the data in a message. Thus, entire processes with all their state and data can be moved to another host by placing their entire address space in a message.

The principal relevance of this system to us, aside from substantiating the claim that message-based distributed operating systems can be successful, is that it demonstrates demand-paging facilities that are network-transparent. This is achieved by allowing memory segments to be defined to be *imaginary segments* that map to a port rather than a disk address. These segments can be mapped into a process's address space just like any other memory segment. However, when a process references a page that is mapped to an pseudo-segment for the first time, the kernel generates a message requesting its contents and sends it to the port associated with the segment. When such a page is about to be purged from main memory, a message is sent to the port containing its new contents. This mechanism allows demand paging across a network so that large messages need not be copied all at once and only those parts of a message need be copied that are actually referenced.

With this capability, one might demand-page the address space of a migrating process as needed rather than having to copy it all over immediately. Although this approach is not suitable for copying address spaces between the old and new hosts of a migrating program because of the residual dependencies it implies, it can be used to advantage if programs are demand-paged from network storage servers.

2.1.4 Other Systems

Remote execution has been an issue in many distributed operating systems, with some also addressing issues of migration. In all cases, the simplifications and restrictions outlined earlier apply to these systems as well. Hence, in relation to the work we present in this thesis, these systems provide no substantial new insights. MOS is another "distributed UNIX" with goals and characteristics that are very similar to those of LOCUS[BL83,Shi83]. SODS is a distributed operating system that relies on a network manager on each machine to locate remote objects and, therefore, also to handle rebinding issues when objects migrate[Ham80,SF80]. Funneling all remote references through a network manager process causes the design to experience performance problems. Amoeba[TM81] is in the process of providing migration facilities.

Several systems provide or plan to provide remote execution of programs, relying on various flavors of message-based communications for all interactions between processes and the provision of a *resource manager* of some sort on each host that controls local resource allocation, including program loading. Systems not based on extensions to some version of the UNIX kernel include RIG[BFL*76,LGFR82], Arachne[Fin80], Charlotte[ACF84],

ArchOs[JP84], and Meglos[GK85]. Systems designed as extensions of UNIX to a distributed environment include Nest[AE85], 4.2 bsd[Uni83], Altos-net[KG83], SDS/NET[AHL*80], Uux[NL80], arid the COCANET system[RB82]. However, with the exceptions of Nest and COCANET, all these systems provide a remote execution environment in which remotely executing programs see the execution environment of the machine on which they are executing instead of the execution environment they would see if they were executing local to the machine of their invoker. These systems, with the exception of Nest, also do not preserve complete UNIX process signaling and process group signaling across machine boundaries, so that remotely executing programs cannot assume the same execution semantics as locally running programs.

A system that requires mention for having been one of the first to support remote sharing of a "pool of processors" is the Cambridge Distributed System[WN80], In this system, each user logs on to a "terminal" host, which requests a processor from the pool of free ones and establishes a reliable byte stream connection to the allocated processor. The user "owns" the remote processor for the duration of his working session. There is no sharing of a processor among more than one user and no migration of programs between processors during execution. Thus, the system supports only remote execution, with file access obtained from a network file system.

2.2 Migration Through Failure Recovery

A number of systems have provided the effect of program migration through failure recovery. When a program fails, either because its host machine crashes or because it has been destroyed (e.g., preempted) for some other reason, the failure recovery facilities restart it from its most recent check point on another machine. The most notable of these systems are the Tandem[Bar81] and Auragen[BBG83] ones.

A similar effect is achieved in the Xerox "Worms" programs through extensive replication[SH80]. These programs create replicas of themselves, called program segments, on any free machines they can find. Loss of a program segment cannot destroy the program unless there are no other segments in existence anymore. Hence, preemption of such programs from a machine simply involves destroying their local segments.

Since our primary goal is access to additional resources rather than fault tolerance, we consider these forms of preemptable remote execution too expensive for our purposes—unless fault tolerance is also a requirement for the system. We will consider environments that are relatively fault-free and do not require fault-tolerant program execution in general, implying that better use of the resources needed for replication/checkpointing of program state can be made by running additional applications instead. For such environments, we view fault tolerance as a separate issue that should be addressed independently from the form of preemption we are seeking.

2*3 Resource Sharing and Security

On top of the basic mechanisms of remote execution and migration, one needs a means of ensuring the integrity of shared machines. Otherwise, users will be unwilling to accept guest programs on tneir machines for fear of potential corruption.

Dannenberg describes an approach to sharing of personal workstations that is oriented towards providing a maximum amount of autonomy, security, and protection to each user[Dan82]. This is achieved by allowing the owner of a workstation to completely define the sharing policies that govern how the host's resources may be utilized by means of a *policy database.* Enforcement of these policies is achieved by means of a *Butler* process,

which controls and supervises all interaction between the host and its users.

The Butler serves as a manager for its local host (managing both local and guest programs) and also as an agent for requests to remote hosts. Execution of a program on a remote host is achieved by having the local Butler (acting as one's agent) specify the program in terms of a *resource configuration* and negotiate with the remote host's Butler (acting as its manager) to have the program executed. If the request is compatible with the remote host's sharing policies, then the remote host's Butler will initiate the program and return the results.

If a program exceeds its resource limits or the sharing policy is changed, then the Butler will take whatever actions are necessary to bring the program in line. This may include warnings, "deportation", or destruction. In the case of deportation, which is the action of interest here, the Butler negotiates with the Butlers of other hosts to find a new host to which the program can be transparently migrated.

Resource management for a host is performed by its *Banker* process, which keeps track of all resource allocations. This allows all resource revocation or rebinding to be handled in a centralized manner so that the Butler can easily implement its enforcement of host usage policies. Furthermore, all resource references are done by means of (port) capabilities, so that there are no hidden references to deal with.

Dannenburg's work provides a bridge between our work and the topic of policy issues for preemptable remote execution. He formalizes the idea of interacting autonomous hosts and the notion of preemption. His approach of centralizing control and accounting into two distinct entities on each host demonstrates a means by which policies for when to migrate a program and how to find its resource utilization can be implemented. As such, we will use his design as the target model that our preemptable remote execution facilities will support. An interesting result of this is that we are immediately forced to address the issue of machine-relative servers—namely Butler and Banker processes—which others have ignored.

2.4 Global Resource Scheduling

Most of the work that has been done in this area is on theoretical models for load sharing algorithms. These have assumed very simple communications models, frequently with zero communication cost. Little work has been done on determining how realistic such models are; that is, on *how* to actually gather global state information about a system. What has been done, with a few exceptions, assumes point-to-point communications, ignoring the availability of efficient broadcast and multicast support in many local network technologies.

Also, much of the scheduling literature is not directly relevant to us because it is aimed at environments which differ significantly in their nature from that of a workstation-based system of autonomous machines. The following subsection provides an overview of the theoretical work that has been done on load sharing algorithms. A more detailed discussion of the more relevant efforts follows in subsequent subsections.

2.4.1 Overview of Load Sharing Work

An excellent literature survey of this topic is provided in the appendix of Wang and Morris[WM85]. This survey is paraphrased here and comments are made regarding the relevance of various works to our own. Wang and Morris divide load sharing algorithms into six classes:

Hierarchical: Schedulers are organized into a logical control hierarchy where each scheduler is responsible for either lower level schedulers or servers. Load distribution is done by requesting and reserving resources up and down the hierarchy until sufficient

resources are obtained. Load sharing has not specifically been addressed in these schemes. Examples of this type of scheme are described in [Kie81], [WvT80,vTW81], and [MP81].

Sequential: Nodes are arranged in a virtual ring and pass a token around to determine access to shared resources. These schemes emphasize concurrency control rather than load sharing. Examples are described in [CMB*79] and [Lan81].

Diffusion: Load sharing is accomplished by migrating load to neighboring nodes. One advantage of this is that communication cost can remain relatively low even when system size increases. However, the rate of diffusion and the stability of the system are not fully understood. For distributed applications, which can generate a large number of subprograms in a short period of time from a single source (e.g., parallel compilation of separate program modules), diffusion becomes an especially questionable approach. Systems that have employed diffusion approaches include MuNet[HW80], CHoPP[SBK77], and Arachne[BF81].

Contract bidding: Sources seeking to distribute their load broadcast a "request for bids" to all servers. Servers submit bids (or don't, as the case may be) according to local load criteria, such as job queue length or processor utilization. In our environment, they would also take into consideration local policy considerations concerning availability. Later, the requesting source chooses a server on the basis of the best bid received. A reverse form of this scheme, in which servers broadcast requests for work, can also be constructed. This class of algorithms depends heavily on the appropriateness of the choice of bid and on communication overhead. Except for work presented in [SG85], these issues have not been studied much. DCS[FL72] and CNET[Smi79,Smi80] are two systems that have employed bidding schemes to perform global scheduling.

State feedback: Estimates of the global system state, such as processor utilization values or job queue lengths, are maintained by each node and periodically updated. Sources simply route work to the "least loaded host". While these algorithms provide the best performance under trivial communication cost assumptions, they have problems when communication overhead and staleness of state information are taken into account. Much of our work will center around an examination of how to manage these problems in our particular type of environment.

Dipstick: Each node only executes tasks remotely if its local load is above some "cutoff" value. Early work, e.g., [PD80], [SF79], [How72], and [Cas81], with this type of scheme exhibited stability problems with respect to the choice of load cutoff value: Provisions must be made to avoid hosts spending all their time shuttling jobs among themselves and never executing any. New research by Eager et al. [ELZ84], indicates that a surprisingly simple form of this approach works very well in conjunction with an appropriate location algorithm for routing tasks to be executed remotely. (This work is discussed in more detail later on.) Also note that this approach can be combined with any of the other schemes described.

As mentioned earlier, much of this work assumes an environment that is significantly different from ours. Relatively few sources and servers are assumed so that communication costs can be kept low. Our environment assumes a randomly changing set of sources and servers of large size (potentially hundreds of workstations). Schemes that make use of nearest neighbor communication or that order servers in some way, for example into a virtual ring, must now deal with the problem of dynamically updating information about who should communicate with whom.

Fairly homogeneous generation of tasks is typically assumed, by which we mean that most sources are assumed to generate tasks and sources are assumed to generate tasks at roughly the same rate, averaged over time. Also, the number of tasks generated is

generally assumed to be large relative to the number of sources and servers. Consequently, task queues at sources and/or servers are typically non-empty. In contrast, our workload assumptions imply that many potential servers will be idle and that programs to execute remotely may be generated in a bursty manner. Schemes that rely on cyclically querying all sources or servers may waste considerable effort querying idle sources or unavailable servers.[2] In the mean time, a source that suddenly generates several programs for execution may have to wait before it obtains any service at all. Schemes such as diffusion must also deal with this problem of "hot spots", as mentioned earlier.

Another assumption frequently made is that all of the workload in a system is processed by its servers. If servers query idle sources, they are only wasting their own resources. In our environment, personal workstations may have significant locally generated loads in the form of interactive applications that are not part of the servers' workload. Servers looking for work should not bother these potential sources since they are busy with their primary purpose of providing local service to their owners. The determination of the utility of server-initiative schemes must take this communication overhead into account.

An interesting side result mentioned by Wang and Morris is that any scheme that manages to keep all available servers busy will yield the same behavior with respect to average response time to finish a task. Furthermore, for server utilizations up to about 70%, these schemes are essentially as good as those that use considerably more information than is usually available, e.g., "service shortest task first" schemes. This implies that any algorithm, such as "send to the least loaded host", can be expected to perform almost optimally under most circumstances as long as its communication costs can somehow be contained.

2.4.2 Random Probe Sets

An even stronger statement on how well relatively simple load sharing schemes can be expected to perform is presented by Eager *et al.* [ELZ84] This paper presents evidence that extremely simple dynamic load sharing policies, which collect only small amounts of global system state information, yield almost optimal scheduling results. Optimal scheduling is defined to be an $M/M/K$ server queue, where K is the number of servers[Kle75]. That is, optimal performance is modeled as a K-multiprocessor that always is able to assign the next queued task to any available processor. Based on this result, they argue that simple policies offer the greatest promise in practice, because of their combination of nearly optimal performance and inherent stability in the face of both scalability and inaccurate state information.

The algorithm they present as a good candidate for implementation is one based on *random probe sets* and a *threshold* load value for determining whether to execute a task locally or remotely. In Wang and Morris' classification, it is a combination of a state feedback and a dipstick approach. Each server keeps track of a single-valued load metric for itself, such as the job queue length. When a new task is to be executed, it is executed locally if the load metric is below the threshold value. Otherwise, a remote server is selected, which then performs the same threshold decision. Tasks are allowed to be "forwarded" at most some fixed number of times to prevent the system from thrashing (which one can prove that it will inevitably do). Remote servers are chosen by probing a small random set of servers in the system for their loads. The least loaded machine is selected from this *probe set.*

Using queueing models and simulation, this simple scheme is shown to perform almost optimally. To reflect the existence of the minimum communication cost incurred even in the optimal case, service times are weighted by a factor corresponding to the communication overhead needed for the theoretical minimum number of task transfers between machines

[2]One of the results that Wang and Morris present in their paper is that cyclic scheduling schemes of this sort work surprisingly well.

if the queueing server were implemented with each server on a separate machine.[3] The results also indicate that a very simple threshold policy is satisfactory and that increasing the probe set size beyond about 5 only helps at very high loads. The recommended values are a threshold value such as job queue length of 1 (i.e., execute remotely unless there are no local tasks running) and probe set size of 5.[4]

The advantages of this algorithm are clear: Bounded, small communication costs, provable stability, and excellent performance under the environmental assumptions made. Unfortunately, the algorithm relies on homogeneity and statistical assumptions similar to those outlined earlier, which we have argued don't necessarily hold in our environment. That is, it relies on all or most hosts having similar enough loads to allow the system's state to be fairly accurately characterized by a small random sample of them. The degree of accuracy needed depends in part on the frequency with which programs are remotely executed: The higher the frequency, the shorter the time that a lightly loaded host will have to wait before being found by some probe. Also, if *all* of the system's load is scheduled using this algorithm, then the system load will automatically be distributed in a homogeneous manner since the algorithm (by definition) tries to balance the load evenly across machines.

In contrast, consider a case in our environment where only a few machines are generating new programs for remote execution, many machines in the system have high loads due to locally generated programs, and a cluster of idle machines exists in one part of the system. The machines desiring to remotely execute programs will only rarely find an idle machine because their random probes will almost always find only the heavily loaded machines, even though there are sufficient idle machines in the system for their needs. Thus, while we have argued that, in general, we expect there to be many idle machines available, in which case almost any scheduling algorithm should work well, the Random Probe Set approach will probably not perform well for the various "non-standard" situations that can occur in a workstation-based environment.

2.4.3 Market Approaches

An interesting approach to the problem of fair resource allocation is a market-based one, such as that proposed by Malone[MFH83]. In a market approach, allocation is performed by defining a monetary currency and treating resources as goods to be bought and sold. Hosts sell the use of their resources to clients for whatever price they think the market will bear. In this manner, scarce resources will be allocated to high priority jobs that are willing to pay for them. Similarly, when resources are plentiful, they will have to be offered at a low price, allowing applications to acquire a large number for a relatively low cost. The principal difference between this approach and the usual contract bidding schemes lies in the determination of appropriate market prices. Most bidding schemes employ fixed pricing schemes.

Unfortunately, this kind of approach also does not seem to fit well into our environment. Since our workload is variable and bursty in nature, the price of resources on any given host will have to fluctuate rapidly and over a large range of values. It is not clear exactly how hosts should set their pricing policies under such conditions.

Preemption causes additional problems: Since local users have priority access to their personal workstations, they effectively have the right to "break contracts" made by the workstation in their absence concerning resource allocation. One might consider attaching a monetary cost to the act of preemption, but it is not clear what value should be used.

Given these circumstances and the fact that we do not prioritize jobs with respect to each other, we have chosen not to pursue this avenue of investigation. However, it would

[3]The minimum number of transfers is derived in [LM82].

[4]A threshold of 2 is recommended when server utilizations rise above 70% or so.

be relatively easy to supplement our scheduling facilities with a pricing scheme, so that one might consider doing so if the issue of prioritized applications becomes important.

2.4.4 Variable Supply Model

Singh and Genesereth describe a different extension of bidding schemes that explicitly takes communication costs into account and makes use of a broadcast communication medium[SG85]. By suitable monitoring of broadcast messages and an assumption of idempotent task execution, they are able to eliminate the need for *award* messages in the standard *request-bid-award* sequence. A caveat, however, is that they assume reliable broadcast, which may be too expensive for a large system. While we cannot make their idempotency and reliable communications assumptions, the trick of monitoring traffic to obtain hints about system state is a useful one.

Unfortunately, the means used in this model for communication cost control is based on having a queue of unexecuted tasks at each processor, which executes tasks sequentially. Communication cost is varied by deciding whether tasks are queued at their source or at servers. Tasks queued at servers are queued in a *potential task set* at all servers until some server "grabs" them and announces the fact via broadcast to all others. While this approach is very appropriate to the application domains they are interested in, it is not suitable for our purposes since we do not assume sequential execution of programs and wish to avoid multiple invocations of possibly non-idempotent programs.

2.5 Summary

The relevance of other people's work to this thesis can be summarized as follows:

- Existing distributed operating systems have demonstrated the viability of computation in a distributed environment and the desirability of remote execution and migration of programs. Successful implementation of such facilities has been shown to require, at minimum, that programs communicate with each other only by means of well defined, *controllable* mechanisms, such as messages. By controllable communications, we mean that the operating system knows of, and could therefore redirect, all communications that occur between any two processes in the system.
- Existing systems and designs have made simplifying assumptions. Most notably, they have ignored the fact that a program may have machine-relative state stored in places other than the kernel of a machine's local operating system. As a result, migration of a program may cause it to have residual dependencies on the machine-relative servers of the host it migrated from. They have also ignored the question of interference between migration and the normal operation of the system as a whole. Many designs are also either fault-intolerant or rely on failure recovery facilities we are not willing to assume.
- The issues of security and resource sharing policies must be addressed once the basic facilities of preemptable remote execution have been provided. Dannenberg has addressed these issues already and we make use of his results and recommendations.
- Most of the work done in global resource scheduling has centered on load sharing algorithms aimed at an environment that is different from ours in significant ways. Its utility for our environment is contingent on successful resolution of communication cost and homogeneity issues.
- Recent work indicates that surprisingly simple load sharing schemes may be the most successful ones in practice because of near-optimal performance and desirable features such as low communication costs and stability. However, its applicability to

our environment still depends on relaxing its homogeneity assumptions.

Chapter 3

Remote Execution

3*1 Introduction

This chapter will cover the issues surrounding remote execution of programs *without* the ability to preemptively remove them by means other than destruction. The advantage of considering just remote execution alone is that it is a much easier problem to solve. Operating systems that employ only network-transparent communication between processes can be modified in relatively straight-forward ways to support remote execution of programs.

"Basic" remote execution achieves our principal goal of providing access to greater numbers of resources than are locally available and is quite useful despite the fact that users either risk losing work when their remotely executing programs are destructively preempted or must give up total autonomy of their machines. If machines provide priority scheduling whereby locally initiated programs receive priority over remotely initiated guest programs, then users may frequently be willing to tolerate guest programs executing concurrently with their own work. Text editing, for example, does not require many CPU resources and hence would not be bothered significantly by the presence of concurrent guest programs. This is especially true when programs are fairly short in duration so that users need tolerate guest programs only for short periods of time before their machine is "cleared" of them.[1] If resources are frequently idle, then destructive preemption of programs will not cause much loss of work in any case.

3*2 Network-transparent Execution Environments

Chapter 1 introduced our definition of transparent remote execution and described how it can be achieved through provision of a network-transparent execution environment for programs.[2] Several existing systems and system designs already support network-transparency, to some degree, by providing access to objects throughout a system by means of communications facilities that are based on globally unique identifiers for objects and whose invocation semantics are independent of location. However, as was pointed out in Chapter 2, provision of such communications facilities does not provide a completely network-transparent execution environment. In this section, we discuss one of the issues that must still be dealt with, namely the need for multiple naming contexts and a correct

[1] An example of an application that might generate many concurrent subprograms of short duration is a compiler that compiles (small) program modules in parallel.

[2] Remember that while a completely network-transparent execution environment is not necessary for the provision of our form of transparent remote execution, we have decided to aim for such an environment, realizing that the network-transparency of some operations provided by a system is not required. See Section 1.2.3 for a more complete discussion of this point.

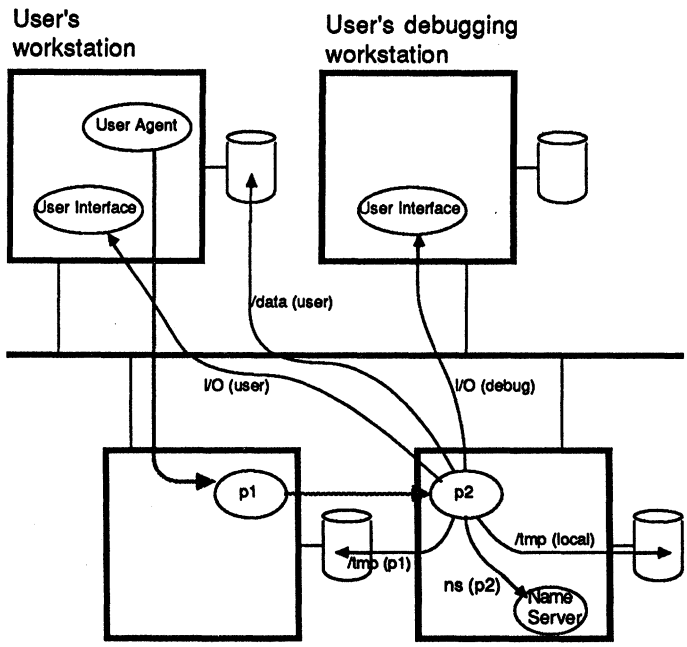

Figure 3.1: Examples of name contexts.

closure of bindings from names to globally unique identifiers for both global and machine-relative services/objects through these naming contexts. The term *name* here is meant to refer to user-assigned character-string names, such as file names, user account names, mailbox names, machine names, and service names. The term *globally unique identifier* is meant to refer to operating system names, such as process identifiers and open file identifiers. The examples given below describe the types of contexts needed to support remote execution and discuss the main issues one must deal with when implementing a correct closure of a set of naming contexts for them.[3] Figure 3.1 illustrates these examples.

The canonical example of an ambiguous relative name is that used to refer to a name server itself. If we assume, for example, that each machine has a local name server, then name service can be provided to a program by the server on the user's machine and/or the one on the machine running the program. If we use just the user's name server, then the name server cannot be used to resolve local names for such things as temporary storage. But if we use just the name server on the machine running the program, then we must copy any predefined name bindings to this server as part of program initialization (e.g., "environment variables" that define such things as the current working directory). Furthermore, the bindings must be kept in a separate context so that they don't conflict with any bindings already in the server, and this context and its bindings must be explicitly deallocated when the associated program exits.

A less obvious example is that of multiple I/O devices. Many operating systems support the concept of "standard I/O", which represents predefined bindings of well-defined names for input and output byte streams to specific device or file instances. These bindings can be viewed as being relative to a context defined as the "initial program invoker's context". But many workstations today have more I/O devices than just a keyboard and a character-

[3]A general discussion of how to implement arbitrary name binding closures is given in [Sal79].

oriented display terminal. Frequently, they also have input devices such a mice and output devices such as one or more graphics frame buffers. All these devices are also part of "standard I/O" and references to them must be interpreted in a context that is relative to the machine at which the user who ultimately invoked the program is sitting.

This problem extends to multiple I/O contexts. Consider, for example, that programs may also want a "debug I/O" context in addition to the one defined for "standard I/O". For each possible I/O path supported as part of "standard I/O", we will want to define a debugging equivalent so that debugging can be decoupled from the program's normal I/O. For example, if users employ multiple "virtual terminals" to interact with the system, then they may well wish to bring up a new virtual terminal to debug a program, complete with separate keyboard, mouse, and (multiple) graphics output instances. Each of these must be correctly directed to the appropriate place, *which may not be on the same machine as standard I/O*. Debugging a program from a machine separate from that to which I/O is normally directed is frequently useful.

Another context that is required is that of the *immediate* invoker of a program. Since (remote) programs can recursively invoke programs, we must be able to refer to the context of the immediate invoker in order to reference such things as the temporary storage server that the invoker was using. Consider, for example, that when creating a new temporary file, we want to create it on the local temporary storage of the machine where the program is running if the file will only be used by the program itself. But if programs create temporary files that will be passed on to other programs, then these other programs must have some way of finding the temporary storage server that was used by the invoking program. While this is not a problem if global, absolute filenames are used, if well-known relative names are used then they must be correctly bound to the first program's definition of them.

One might be tempted to solve this problem by resolving such names before they are passed to a new program. Unfortunately, not all relative bindings may be resolved at the time that they are passed to a new program or obtained from a name server. Consider, for example, the idea of a null output device, to be used for throwing away unwanted output from background programs. For reasons of efficiency *and* reliability we want to implement this facility local to the machine on which a program runs. But if the operating system allows programs to pass in absolute identifiers rather than uninterpreted names (as UNIX does, for example), then there is no opportunity for the program or a name server to redirect an I/O instance to the appropriate local instantiation of it. We are thus faced with either prohibiting the passing of interpreted names to programs or of providing a lower level mechanism for directing "absolute" references to machine-relative objects.

The former approach requires no modifications to the local operating system, requiring instead that all applications software be changed to use only uninterpreted names. The alternative, if allowable, is to change the manner in which the operating system implements the mapping of absolute references. Essentially, one needs the concept of *well-known machine-relative identifiers*: The operating system must be able to recognize certain identifiers as referring to machine-relative entities and must map them accordingly each time they are used. Of course, this implies that there must be a mechanism for allowing local servers to register themselves with the operating system as implementing a particular well-known service or object. Chapter 7 describes an implementation of these concepts based on a generalization of references to *process groups* that has been developed by Cheriton and Zwaenepoel[CZ85]. The implementation demonstrates that only a very slight penalty in performance need be incurred when referencing these "special" objects via this more general identifier type.

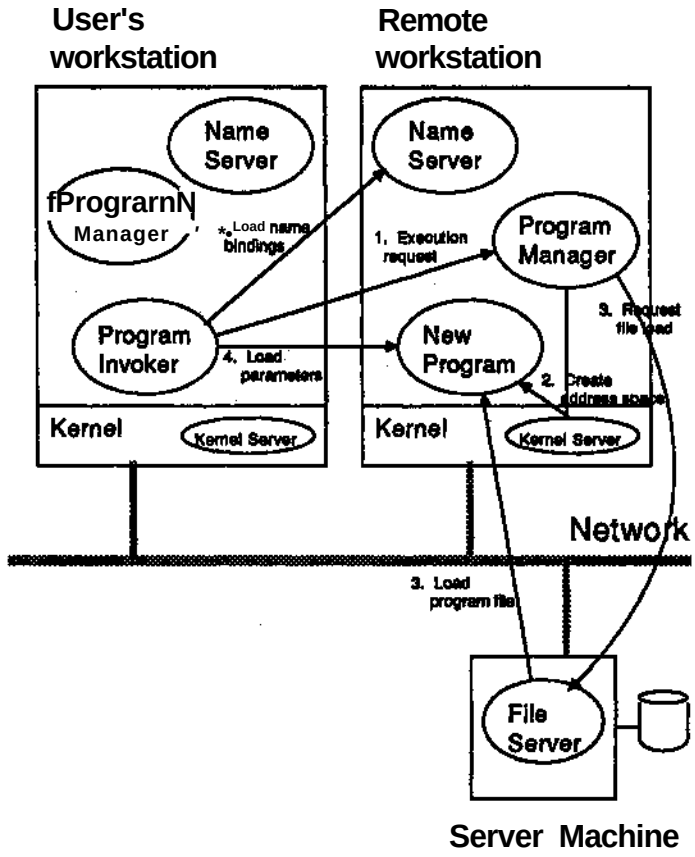

Figure 3.2: Communication paths for program creation with remote loading.

3,3 Instantiating Programs

One of the most important aspects of remote execution that must be dealt with properly is program instantiation: Finding the correct program binary image to load, loading it onto a particular machine, and passing in appropriate parameters such as command line arguments and environment variables. As discussed in the previous section, the crucial issue is one of binding references to the correct servers and to the correct objects within those servers, and being able to communicate information in a network-transparent manner. For example, the file server used to supply a program load image cannot assume that the address space to load into will be locally accessible—it must be able to load programs across the network or talk to an intermediary on the remote machine that will do the actual address space manipulations for it. Thus, if there are operations, such as address space creation, that must be invoked locally, then the operating system and program initialization must be structured so that a local "agent" exists on each machine that performs all necessary local operations. Consequently, this local agent requirement also implies that local execution of a program is just a special case of program execution on any machine.

As an example, consider the following outline of an implementation for program initialization (the details of this implementation are given in Chapter 7.). Figures 3.2 and 3.3 show an example of program instantiation using a network file server for loading of the program binary image and separate server processes for various operating system functions. Figure 3.2 shows how things work if the address space of the newly created, but still inactive, program can be accessed directly by remote processes. Figure 3.3 illustrates the same example assuming that the new address space can only be accessed locally. Ini-

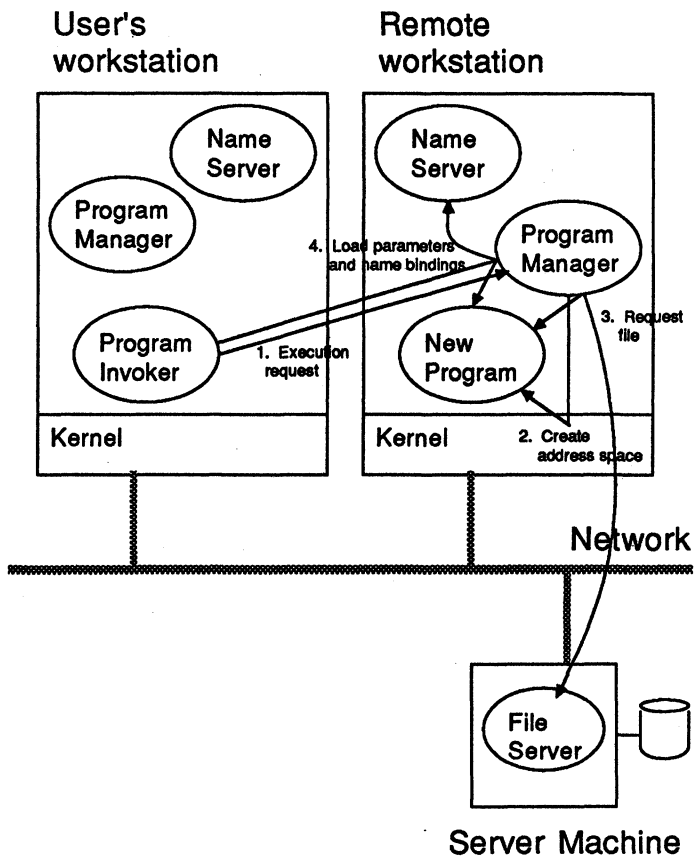

Figure 3.3: Communication paths for program creation using a local agent to perform all loading.

tiating execution involves an invoker, such as a command interpreter, sending a request to the appropriate machine's local agent, called a program manager here, to create a new program instance (step 1). The program manager invokes a kernel operation to set up an address space and process descriptor (step 2). This is frequently an operation that must be invoked locally. Depending on whether address spaces can be loaded from remote file servers or not, the program manager may act as an intermediary for program loading or may turn over control of the newly created address space to the network file server for direct loading of the program image (step 3). Next, the program's parameters must be initialized (step 4). If this can only be done as a local operation, then the invoker must send the program manager the parameter definitions so that it can place them appropriately, which may involve initializing local servers as well as the program's address space. If parameter initialization can be performed with network-transparent IPC, then the invoker can be allowed to initialize parameters without having to go indirectly through the program manager. The final step is to set the program running, which can again be done either by the invoker or the program manager, depending on locality constraints and operating system design. (For example, in the implementation described in Chapter 7 the invoker controls when a program starts running so that operations such as remote debugging can be performed.)

The only difference between local and remote program execution in the example just presented is that the initial load request is sent to a different program manager. Notice however, that if remote execution did not need to be supported, then the design could be simplified to remove the program manager since the invoker can just as easily perform all local operations himself. Hence, remote execution does not fall naturally out of most operating systems designs, but must be explicitly taken into consideration.

Requiring a local agent on each machine for all program execution requests also fits nicely with the recommendations of Dannenberg for dealing with security and resource sharing among multiple users. The program manager described above serves exactly the functions proposed for Dannenberg's Butler process.

Another aspect of program loading we must cover is that of determining which program executable image to use. There are two interrelated questions to consider: If program invocation involves following a *search path* of possible directories in which to find the correct binary image, we must decide who will perform the search. If some system directories are replicated on several machines, then we must also determine which copy to use. There are three places where the search path can be searched from: At the program invoker, at the file server if it is remote from everything else, and at the program manager of the machine where execution will take place. If there is only one network file server (i.e., no replication of system directories) then it doesn't much matter who performs the search path evaluation, subject to access rights considerations. (This is not so if we take lookup overhead into account. Then it's a tradeoff of how fast the file server machine is.) However, if some machines have local disks containing copies of standard system directories, then care must be taken with the choice of search path evaluator to avoid unexpected adverse effects. We wish to avoid loading a program from a remote file server if a *faster* local one can perform the same task.[4] But if the program invoker does the search, then it will find its local system directory instead of the one on the remote machine. Hence, load requests might define their program file names to mean the "local" copy and pass *uninterpreted* names rather than pointers to opened file instances to whomever is performing the actual loading operation. But this, in turn, implies that the remote program manager must gain access rights to the specified file. Alternatively, the program invoker will have to query the remote machine to determine whether it has a local file server and whether that file server has the requisite files.

[4]Several studies have demonstrated that large, fast network file servers can frequently out-perform local, small file servers or can at the least provide comparable service. See, for example, [LZCZ84].

The final topic we will mention in this section is an alternative approach to name service that is being developed by Mann and Cheriton[CM85]. As discussed earlier, if we assume that name service is provided by name servers that are resident on each machine, then we are faced with several problems:

- We must decide which name server to use for any given name binding. Not only must we decide at design time which name servers should store what information, but depending on design, programs may also have to decide at run time which name server they must use to resolve a given name binding.
- We may have to replicate name bindings in several name servers, implying that we must worry about allocation and deallocation of information in the name servers. Worse yet, if we assume that globally visible changes to name binding information can be made, such as the user changing his current working directory and wanting all running programs to reflect the change, then we must also worry about maintaining consistency among multiple copies of information.

The approach being developed by Mann and Cheriton obviates the need for local name servers on each machine by pushing the implementation of name service into each program and each server that implements named entities. Multicast communication is used to resolve any references that can't be resolved by information passed to a program as part of its initialization. This information corresponds roughly to the environment variables that UNIX passes into programs to perform its name resolution. The benefits for remote execution are clear: The question of which name servers to use and initialize is moot and the question of how to manage replicated naming information is dealt with as part of the normal procedure of creating and destroying programs themselves. The disadvantages of this approach are that servers must now implement name resolution for the objects and services they provide and understand a name resolution communications protocol. Mann and Cheriton argue that the overhead of forcing servers to implement their own naming is minor.

3.4 Device Access

An important implication of our requirement that a program's execution environment be network-transparent is that I/O devices must be accessible remotely. But some devices, such as graphics frame buffers, may not be accessible remotely in an efficient manner; direct remote access may not even be provided by the operating system design. If this is the case, then remote access to such devices requires that they be managed by locally co-resident servers that act as intermediaries.

This in turn implies that programs that rely on *direct* access to such devices cannot be executed remotely from the machine to which the devices they use are attached. Such programs are usually part of single address space systems in which access to the operating system is not necessarily constrained to a formalized interface layer. Operating systems of this type include Pilot[RDH*80], Cedar[Tei84], and those used on most Lisp Machines[Sym82].

The real reason for providing direct access to devices is typically not because the operating system allows it, but because it allows a particular style of I/O interaction that has many desirable features. The advantage of direct access to I/O devices, such as mice and graphics frame buffers, is that it allows applications to interact with human users in an efficient application-specific manner. For example, applications can provide operations such as "rubberbanding" and "dragging" of graphical objects that occur on every mouse movement without incurring the overhead of going through a separate server on each mouse movement. If multiple applications are being presented on the same output devices, then provision of shared "window management" routines and data structures can

be used to provide efficient controlled device-access of each application. For this reason, even some systems that place each application in a separate address space may provide a shared memory segment among address spaces in which a common graphics subsystem resides[CD83].

Various studies have been performed to determine how well loosely coupled I/O connections can be made to support various applications classes. These have shown that high-level network graphics protocols allow a large class of graphics applications to be supported with a general "front-end" server that provides such things as virtual graphics terminals[LN84]. Such servers implement a standard set of graphics operations or objects, depending on the paradigm used for interaction, and applications are thereby able to do I/O using relatively low bandwidth communications.

However, these servers may not be able to provide the kinds of speed and efficiency required by applications that wish to perform highly interactive, application-specific I/O actions that push the limits of the hardware used in order to achieve "acceptable" performance. For such applications, maintaining acceptable performance will require them to be split into local "front-end" and optionally remote "back-end" components that appropriately divide their interactive and computational tasks between themselves and communicate in a more loosely coupled fashion.

In conclusion, some programs may have to be restructured to employ a more loosely coupled I/O paradigm in order to maintain a performance level that is acceptable to them when remotely executed. This restriction should not affect users too much since it appears that many, if not most, applications can be handled within this restriction without too much difficulty. However, the restriction does imply that converting operating systems and applications that depend on the performance characteristics of a shared-memory-based system and/or programs being responsible for directly manipulating their own I/O interactions will be difficult to extend into an environment which supports remote execution.

3.5 Program Invocation

The existence of programs that users may wish to run strictly local to their own machine—for example, because they are highly interactive in nature or because they require additional security precautions—implies that the semantics of program invocation must include explicit designation of whether a program must be run locally or may be run anywhere. Either a program's binary load image (which is the only object that the invocation facilities have access to) contains locality information or the invoking agent must supply that information.[5] If the program's load image is to contain the information, then we are faced with requiring that all application programs, or at least all those that must be executed locally, be modified to include the requisite information. If we are not willing to require that application programs take into account this new interface to the operating system, then we must require that locality requirements be passed as an explicit parameter to program invocation. That is, users will have to explicitly designate whether programs are to be executed locally or remotely and program libraries will have to include an explicit parameter for such information.

A third, more exotic approach one might consider is to somehow monitor the network traffic of programs and migrate them between machines according to some decision criterion based on migration overhead, machine loads, and network traffic load. However, since it is response time to the end-user that we care about in interactive I/O and not throughput,

[5]One might also place this information in a separate directory entry or file descriptor that is associated with the executable program image file, but this really just extends the definition of executable program image to include two entities instead of one.

it is not clear how one could distinguish between cases where high traffic activity merely indicates inherently high throughput requirements that are independent of locality (e.g., writing a file) and cases where it indicates response-time-degrading interactive activities. One might consider having the operating system of a machine monitor the activity of its devices as a function of local/remote access and attempt to migrate programs accordingly, but such considerations are beyond the scope of this thesis.

3.6 Summary and Conclusions

In this chapter, we have covered issues relating to the provision of remote execution *without* the ability to preempt programs by means other than destruction. We claim that even this "basic" form of remote execution is quite useful, especially if there are frequently many idle resources and if guest programs are typically short in duration. If nothing else, remote execution provides a convenient way of instantiating distributed applications on a set of "reserved" machines. The big advantage of providing just remote execution is that it is relatively easy to graft on to many existing operating systems that already provide network-transparent communication between processes. The requirements described in this chapter can be implemented, for the most part, by adding server processes to the system and modifying the manner in which user-level programs receive parameters and environment information. In contrast, the requirements described in Chapter 4 for support of migration involve significant changes to existing pieces of the basic operating system, such as its kernel and file system.

An interesting implication of providing just remote execution is that it favors a programming paradigm that generates many short programs rather than fewer programs of larger size and longer running time. Since long remotely executing programs risk a greater chance of being destroyed, there is an incentive to split them into several components, even if the overhead of loading several program images would otherwise argue against fragmentation.[6] The availability of preemptive scheduling that allows local programs to obtain service before guest programs reduces the problem of destructive preemption, but cannot solve it since there will still be contention for resources such as memory or overhead due to paging. Also, many users are simply unwilling to share their machine with guest programs while they themselves are using their machine, despite any security assurances the operating system might offer them (see Chapter 7).

In this chapter, we have discussed various requirements that an operating system design must satisfy in order to provide a network-transparent execution environment. As discussed in Section 1.2.3, provision of a network-transparent execution environment guarantees the form of transparent remote execution we seek.

An important form of functionality needed to support network transparent execution environments is the provision of a concept of (logical) locality with respect to various entities, such as the invoker of a program or the machine on which a program is actually executing. This form of locality must be provided as part of the semantics of naming because various services and objects are implicitly bound together by their common physical location, for example, all I/O devices of the machine that a human user is sitting in front of, or the program manager of a given program, wherever that may program may be. Note, however, that the form of locality needed must not depend on the specific physical location of the objects being grouped together by location.

A second requirement is that all operations that the operating system can only perform locally must be "fronted" by a local agent. This typically includes various aspects of

[6]Chapter 8 briefly discusses an alternative approach based on making programs fault tolerant. Splitting programs into several subprograms is in many ways equivalent to performing checkpoints on their state at the split points.

program instantiation and access to hardware devices. Also, since machines may wish to control the use of their resources, a local server may need to be provided on each machine that controls acceptance of remote execution requests and acts as a local agent for such requests and their subsequent (local) resource requests. If local execution requests also go through this server, then local execution becomes merely a special case of program execution on any machine.

Note that the number of operations that must be executed locally depends on the design and implementation of the operating system being used. In principle, one could implement all operations such that they can be executed in a network-transparent manner and could place the policy database that determines access rights into the kernel of the operating system as well. However, the approach of using local agents allows remote execution to be added to a system in many cases, even if it was not designed to be completely network-transparent in its operations to begin with.

When access to hardware devices may be inefficient or even impossible from remote programs, remotely executing programs must funnel their interactions with such devices through local agent processes that reside on the same machine as the devices they manage. While a great many applications can be handled in this manner, either because they need no more than simple byte stream connections or because they are able to communicate their I/O requirements to a local agent by means of a high-level communications protocol, there are some highly interactive applications that cannot be serviced in this manner without incurring unacceptable degradation in performance. Such applications, which require a tight coupling of some of their state to the hardware devices they wish to use, must be split into local and remote components or must be executed strictly local to the machine to which those devices are attached. Thus, a true distributed time-sharing system in which every program may be executed anywhere may not be practical. In this case, a third requirement must be imposed on the operating system: The program invocation facilities used must somehow be able to obtain information about the locality constraints of each program to execute.

Chapter 4

Preemption and Migration

4.1 Introduction

This chapter deals with the issues of preemption and migration of remotely executing "guest" programs. One way to preempt guest programs is to simply destroy them. If applications such as compilation and text formatting are idempotent, then destroying them would not cause any serious damage. Applications that cannot tolerate the possibility of destruction need the support of failure recovery facilities in any case. Thus, preemption by destruction would seem to only lose cycles rather than cause irreparable damage. There are two reasons why this approach may be unsatisfactory:

- Since the whole point of remote execution is to gain access to additional computing resources, we must be careful not to lose more work through preemptive destruction than is gained through remote execution.
- Using destruction as the standard means of preemption changes the failure statistics of a system. Facilities to support fault tolerance must rely on assumptions about the failure characteristics of a system for their implementation. Consider, for example, that preemption is likely to occur on many machines almost simultaneously at the beginning of a working day, after lunch, or after a group meeting of several individuals. If fault tolerance is based on the assumption that the probability of more than one machine failing is negligible, then simultaneous preemption on several machines can easily wipe out applications that are supposed to be highly reliable.

The alternative to destructive preemption is to migrate a preempted program to another machine that is willing to accept it. This still leaves the question of who should perform the actual migration procedure. If applications understand that they may be preempted from a host, then they can be structured to perform the process of migration to another machine themselves. However, this would require a major restructuring of most application programs, which we consider to be an unacceptable requirement to impose.[1] The alternative is to have programs be migrated transparently by the operating system. If this can be done without significantly disrupting the system as a result, then it is the preferred form of migration. Transparent program migration is the form of preemption that will be discussed in the remainder of this chapter.

In addition to the preemptive uses of migration, one can also derive various load balancing benefits from its provision. These will be presented in this and subsequent chapters and are briefly summarized here:

[1] **LOCUS employs an intermediate version of this approach by requiring applications to define a migration signal handler. However, the actual mechanics of migration are provided as a system call that is implemented by the operating system.**

- Preemptive migration provides a form of load balancing that prevents resource contention between guest programs and significant local activities. Thus, longer-running programs can maintain access to a reasonable share of resources rather than being "starved" by privileged local activities.
- Migration allows us to run global servers, such as authentication servers and configuration monitoring servers, on general-purpose machines rather than dedicated server machines.
- Migration allows us to implement a load balancing scheme that, together with an appropriate global scheduling policy for host selection, provides approximately equal CPU shares to the subprograms of distributed applications whose subprograms are independent of each other. This is important for applications such as those that run at the speed of their slowest subprograms.

Note that while these examples represent specific instances in which load balancing/sharing are useful, we will not attempt to address issues of the general utility of load balancing facilities in this thesis.

4.2 Overview of Migration Issues

The principal problem with migrating a program is the interference it causes with the execution of the program and the functioning of the rest of the system. Not only must the program be atomically transferred from one machine to another so that a globally consistent view of the migration is maintained, but we must also decide how to deal with any *residual dependencies* that the program might have on the previous machine. Residual dependencies are defined here to mean dependencies on state that is maintained in servers that reside on the machine from which a program has been migrated. The two issues interact with each other in that the removal of residual dependencies requires migrating state information that resides in various machine-relative servers.

In addition to the requirement of atomic transfer, migration must also satisfy certain performance constraints. Since execution of various parts of the system must be suspended and various interactions must be delayed, we must worry about global inconsistencies caused by timeouts occurring. For example, a process expecting a reply message from a migrating program may conclude that the program has expired if the reply is not forthcoming within a certain amount of (real) time.

Even if we design the system so that timeouts of this sort can't happen, which is quite difficult given that many applications implicitly maintain some model of temporal behavior, migration may still cause global degradation of system performance. The simplest example is a central system server that must wait upon completion of an interaction, such as moving a segment of data into its address space, with a migrating client. If the server is single-threaded, then the delay caused by this interaction will delay service for all other clients as well. Although a simple restructuring of the server can solve the difficulties of this particular example, the problem of unexpected long delays is a general one of properly dealing with flow control. Similarly, performance may be degraded if suspension of some activities causes repeated retransmission of possibly large amounts of data in messages.

Even if we are able to migrate a program in a consistent and sufficiently quick manner, we are still faced with migrating any associated state that may reside in machine-relative servers, as mentioned above. Some of this state, such as that in the program manager, must be migrated for correct execution. Other state, e.g., local temporary files, need not be migrated for correct operation since it can still be accessed remotely. But if this state is not migrated, then the migrated program will continue to impose a load on its previous host, thus diminishing the benefits of migrating the program. Furthermore, a failure or reboot of the previous host would cause the program to fail because of these inter-host

dependencies. However, some of this state may be difficult to migrate. For example, local temporary files may be quite large and their migration may create name conflicts. Section 4.4 discusses the associated problems in detail.

4.3 Implementing Program Migration

This section will deal with the mechanics of migrating a program's "local" state. By local state, we mean the state in one or more address spaces that comprise the program and in the kernel of the operating system.[2] State kept in various machine-relative servers will be discussed in Section 4.4.

The simplest approach to migrating a program is to freeze its state while the migration is in progress. By *freezing* the state, we mean that execution of processes in the program is suspended and all external interactions with those processes are deferred. We have already explained why the long interaction delays caused as a result are unacceptable. The effect of these problems can be reduced by copying the bulk of the program state before freezing it, thereby reducing the time during which it is frozen. We refer to this operation as *pre-copying*. Note that while pre-copying reduces the freeze time of a program, it does *not* reduce the time required to preempt the program from its original host. Thus, the complete procedure to migrate a program is:

1. Locate another machine that is willing and able to accommodate the program to be migrated.
2. Initialize the new host to accept the program.
3. Pre-copy the state of the program.
4. Freeze the program and complete the copy of its state.
5. Unfreeze the new copy, delete the old copy, and rebind references.

The first step of migration is accomplished by the same mechanisms employed to find a host candidate for remote execution. These mechanisms are discussed in Chapter 5. The remainder of this section discusses the remaining steps.

4.3.1 Initialization on the New Host

Once a new host has been located, it must be initialized with descriptors for the new copy of the program. To distinguish between the *operation* of copying data and the old and new copies of the program themselves, we will henceforth refer to the copies of the program as old and new *instances* of the program. In order to copy the state of the program, we must be able to explicitly refer to both the old and the new instance of the program at some level of the operating system.

Distinct identifiers can be provided either at the user level or internal to the operating system kernel only. In the former approach, we actually obtain two *distinct* entities that are visible to the rest of the system as such. The advantage of this is that much of the migration procedure can be implemented in processes external to the kernel and that copying the bulk of the program state can be performed using the system's standard IPC primitives for data transfer. It also requires that after the program's state has been copied to the new instance we must atomically rename the new instance to be identified as the old one. This must be done *in addition* to rebinding references to the program to find its new location.

[2] We will assume that a program can include more than one address space in order to accommodate designs that wish to ensure that a *set* of programs remain local with respect to each other. For example, a compiler might create a private RAM disk server for its temporary files, which should migrate when it does.

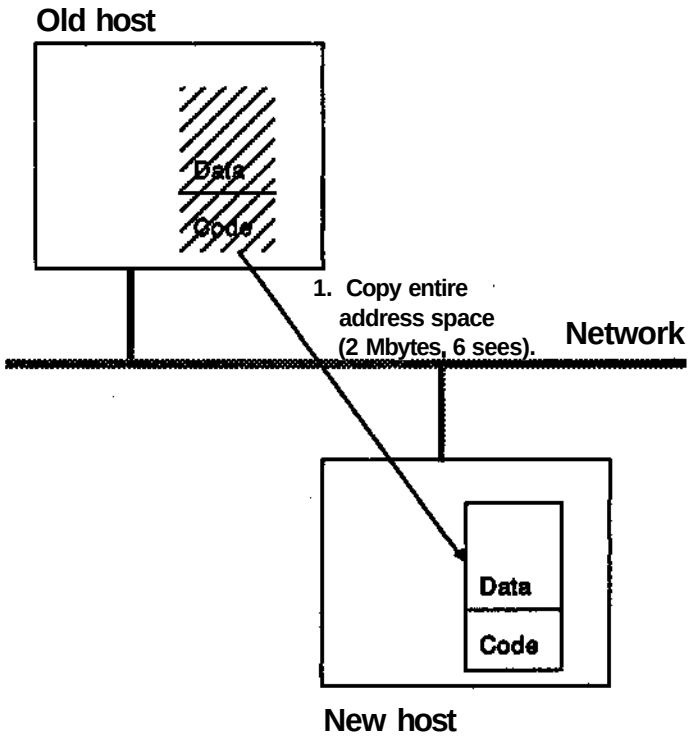

Figure 4.1: Pre-copy an address space. Initial step.

The alternative approach is to hide the difference in identifiers within the kernel. This implies that we need not separately rename the new program instance—rebinding external references to the program will automatically bind them to the new instance. However, it also implies that the bulk of the migration procedure must be implemented within the operating system kernel since there is now no way for external processes to refer to both the old and the new program instances. Aside from forcing a considerable amount of code into the kernel where it can't be jettisoned when not used, this may also cause considerable duplication of code: Many parts of migration can be implemented external to the kernel using the standard system operations already available. Implementing migration strictly within the kernel would require that these standard operations also have an internal invocation interface, which is frequently different enough to be non-trivial to provide. Additionally, the kernel would either have to deal with the performance problems of non-interruptable migratidn operations or with the multiplexing/state-machine problems that are inherent in interruptable operations. Therefore, the approach of making the program's instances visible external to the kernel is the preferable one.

4.3.2 Pre-copying the State

Once the new host is initialized, the state of the migrating program must be copied to the new instance. The objective of pre-copying is to copy those parts of a program's state that are not currently in flux and that represent the bulk of its state before freezing the program. Since the kernel descriptors of a program typically represent only a small amount of data, changes to which are generally expensive to record, there is usually little point in pre-copying them. Pre-copying, therefore, involves primarily the address spaces of the program, and only this form of pre-copying will be discussed here.

Pre-copying of address spaces can be implemented by doing an initial copy of the

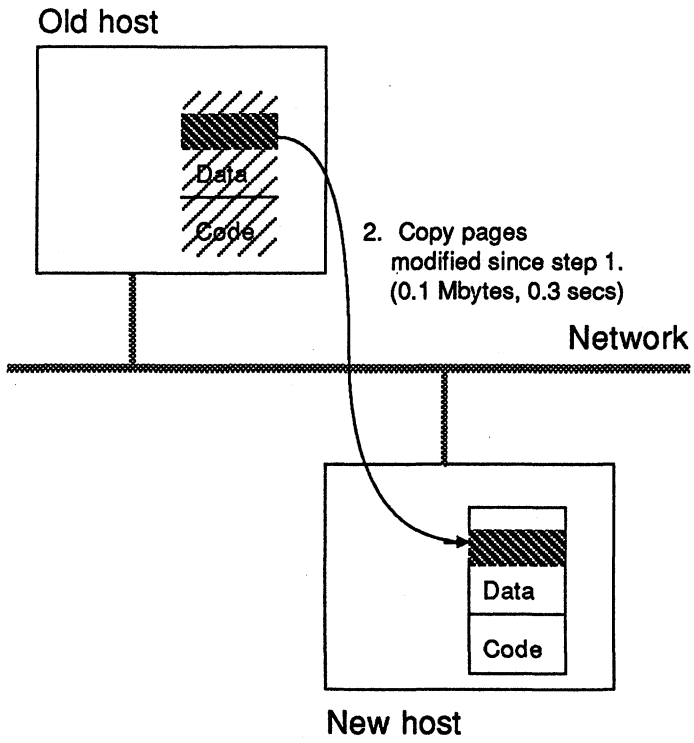

Figure 4.2: Pre-copy an address space. Repetitive step.

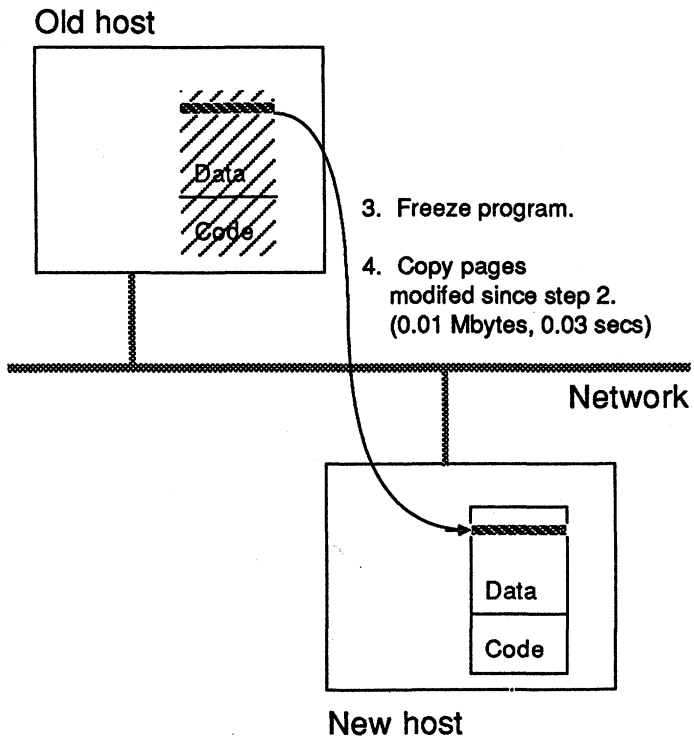

Figure 4.3: Final copy of an address space.

complete address spaces followed by repeated copies of those parts modified during the previous copy. The initial copy operation (which includes copying the code and initial data segments that are never modified) takes the longest time and, therefore, also allows the longest time for additional modifications to the program state to occur. The second copy moves only that state modified during the first copy, taking less time and presumably allowing fewer modifications to occur during its execution time. The third copy should again take less time and allow an even smaller number of modifications, and so on until the "modifications" working set size of the program is encountered. At this point, no more reductions in the "copy set" can be achieved. Consider the following example, depicted in Figures 4.1 through 4.3, which is presented to illustrate both the qualitative *and* the quantitative nature of pre-copying (numbers derived from the implementation described in Chapter 7): Assume that the time to copy a 2 megabyte program, consisting of 1 megabyte of code, 0.25 megabytes of initialized data, and 0.75 megabytes of "active" data, is 6 seconds. If during those 6 seconds 0.1 megabytes of memory are modified, the second copy operation should take 0.3 seconds. If during those 0.3 seconds 0.01 megabytes of memory are modified, then the third copy should take 0.03 seconds. At this point, we might go ahead and freeze the program, having reduced the program's post-freeze copy time from over 6 seconds to about 0.03 seconds. This is a smaller time interval than that used by many systems as the retransmission interval for message packets lost in the network.

Detecting modified state requires that dirty bits be maintained by the hardware. The finest level of granularity is obtained if the hardware and operating system support paged memory. A segmented memory system will not be able to reduce the size of the final copy set to nearly the same extent achievable with a paged memory system. However, even if the hardware does not maintain dirty bits, a significant fraction of a program's state can be pre-copied by simply copying the immutable code and initialized data segments of the program.

In order to ensure termination of the algorithm just described after some small number of steps, we may have to place a maximum limit on the number of copies performed. Otherwise, since a program might reduce its copy set by only a single page at a time, a great number of pre-copying steps could occur if the typical working set of a program contains many pages. Note that programs with inherently large working sets, such as list-processing oriented programs, will terminate the copy iteration as soon as a copy operation does not significantly reduce the "copy set", rather than after needlessly causing the maximum number of iterations. However, the freeze time for such programs will be correspondingly larger than for programs with smaller working sets. One way to reduce this problem is to execute the pre-copy operation at a higher priority than all other programs on the source machine, although one must be careful not to lock out the migrating program from execution and thus destroy the very objective we seek to achieve with pre-copying.

To put the size of typical freeze times into perspective, we point out that a typical operating system operation, such as temporarily swapping a program in a paged virtual memory system out to secondary storage for round-robin scheduling purposes, should suspend the execution of a program for a similar amount of time. If we assume that the system maintains essentially the working set of a program in main memory, then swapping a program to secondary storage will require copying roughly the same amount of state information as must be copied during the final copy step of migration. If we assume that copies of data to secondary storage occur at similar speeds as copies of data across the network, then the swap time for a program should be roughly equivalent to the time needed to make a final copy of a migrating program's state across the network.

4.3.3 Completing the Copy

After the pre-copy phase, the program must be frozen and the copy of its state completed. Freezing the program state, even if for a relatively short time, requires some care. Although

we can suspend execution of all processes within a program, we must still deal with external interactions. When a program is frozen, interactions that can modify its state must be deferred. When the program is unfrozen, these interactions must be forwarded to the new instance of the program for subsequent application, assuming the program was successfully-migrated. (Section 4.5 discusses failure considerations.) Interactions that we must deal with include both direct IPC interactions, such as message sends and replies and data transfer operations, and indirect operations in the form of local kernel operations, such as system calls to change the size of the address space or the priority of a process.

Local kernel operations can be deferred either by having the kernel queue them and then forward them to the kernel of the machine where the new program instance resides, or by having the kernel return operations with a status indicating that they can't be performed at the moment.[3] The latter approach requires that the invoker be responsible for forwarding the operations to a corresponding invoker on the new machine. It also implies that invokers need not worry about unexpected delays that might degrade their performance if they are single-threaded local system servers. However, the former approach has the attraction of being simpler from the invoker's point-of-view and pre-copying can *usually* reduce the delays to less than a network retransmission timeout interval.

Deferral of IPC operations on a frozen program involves both queueing them and forwarding/routing them to the new instance of the program for execution when it has become the valid instance of the program. These operations, which in a message-based system can all be thought of as various kinds of messages, must be queued in a manner that prevents unwanted timeouts, does not overflow the buffering capacity of any machine, and preserves ordering. This can be achieved through appropriate combinations of queueing messages at their source and destination, relying on retransmission of messages, and sending "reset" messages to forestall timeouts.

The simplest approach to deferring messages would be to queue them at their destination, which is the machine on which the old program instance resides, and to forward them to the new program instance's machine when the new instance is ready to accept them. This approach has the problem that queueing large data messages can easily exceed the buffering capacity of the machine. (Remember that we can't copy the data into its intended location within the program's address spaces.) Another small problem is that we must encapsulate forwarded messages in such a way as to designate their correct source location—which is *not* the machine from which the message is being sent out across the network.

The alternative to queueing messages at their destination is to queue them at their source, as depicted in Figure 4.4. This solves the buffering problem since the source machine must already buffer a message. If message semantics are synchronous, then buffering can be done directly in the sender's address space. If they are asynchronous, then the sender will have to be delayed until *after* the receiver is unfrozen again rather than just until the message has been reliably received at the receiver's machine. Fortunately this added delay will be less than a message timeout period in all but a few (rare) cases when the receiver is frozen for a longer period of time. Alternatively, the operating system must be willing to separately buffer messages if it wishes to guarantee bounded return times for invoking asynchronously sent messages.

In any case, we must still deal with the issue of senders timing out and concluding that a program has failed. Note that although timeouts should rarely happen in practice because of the reduced freeze times achieved through pre-copying, we cannot *guarantee* that they won't happen. This requires that the destination kernel respond to the source machine with an indication of what is going on. Since the source machine must retransmit to the destination machine to ensure delivery of a message anyway, there is no point in having a protocol that does more than simply "reset" the timeout counter so that retransmission will continue. One might argue that retransmitting large data messages wastes resources

[3] This is similar to the use of the EINTR return code in some versions of UNIX.

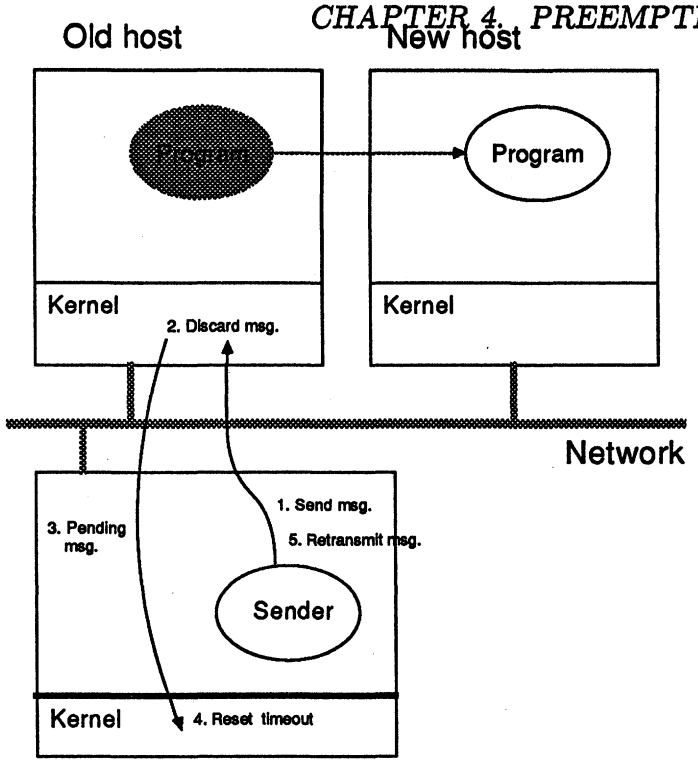

Figure 4.4: Deferring IPC operations by relying on retransmission.

and should be replaced with sending smaller "query" messages. However, if freeze times are typically shorter than the retransmission interval, then the added complexity will only rarely provide any benefit. Another benefit of this approach is that forwarding is now subsumed by retransmission to the new program location once references have been rebound, thereby avoiding the problem of encapsulation.

For local senders, retransmission of the message is meaningless. When the old instance of the program is destroyed, local sends to it must be converted into remote sends to the new instance.

The approach just outlined works for messages that are designed to be sent reliably and hence make use of retransmission techniques. If we must deal with unreliable messages, then the source machine will have deallocated any state and associated buffering for a message long before it receives any indication from the destination machine telling it to do otherwise. Since unreliable semantics are usually employed for performance and efficiency reasons, we cannot expect senders to always wait for an indication from the receiver that no queueing is necessary. Consequently, one might consider simply discarding deferred messages. Unfortunately, this means that any higher level protocols that are built on top of our unreliable messages will have to be prepared to deal with unexpected lossage behavior. Thus, we have merely deferred the problem to a higher, less controllable level of the system. The alternative is to revert to queueing such messages at the destination machine, despite the attendant buffering and source address encapsulation problems this implies.

Another case in which one should avoid simply discarding unreliable messages is when broadcast/multicast communications are supported. If an application employs such communication facilities to send a query message to a group of receivers and expects multiple replies, then migrating it could cause it to lose *all* replies. Migration will thereby have invalidated any statistical assumptions the application might have made to ensure a particular

number of replies.

The last part of copying the original program's state consists of copying its state in the kernel. This involves replacing the kernel state of the new program instance with that of the old one. As part of this, the identifier of the new program instance must be changed to be the same as that of the old one.

Once this operation has succeeded, there exist two frozen identical instances of the program. The rest of the system cannot detect the existence of two instances because operations on both of them are suspended. The instances are essentially "locked" in the same way that a replicated object in a database might be when it is being updated. However, the kernel on the original machine must continue to respond to retransmitted messages in order to prevent timeouts. At this point, we are ready to transfer control to the new instance of the program.

4.3.4 Rebinding References

In order to transfer control to the new instance of the program, we must unfreeze the new instance, delete the old instance, and rebind all references in the system to the old instance to point to the new instance. The first two steps are easy to implement.

Rebinding globally unique IPC identifiers requires, first, that they be mapped through a level of indirection at which the network address to use for sending a message can be changed. We assume that network addresses that are stored at random locations within the state information of a program cannot be found without having application-specific knowledge of their where-abouts. Furthermore, this change must be transparently implementable, implying that the mapping must be done by a system entity, such as the kernel or a network communication server. Systems in which network addresses are embedded in IPC identifiers and systems in which communication can occur in an uncontrolled manner (i.e., no well-known place where references can be remapped) cannot rebind references to migrated programs. Beyond this requirement of controllable indirection, it doesn't matter much how the mapping to network address is implemented or what form of IPC identifier is used. For example, identifiers could consist of local indices into a kernel "port" table that contains the appropriate mapping information for each IPC identifier—as in Accent—or identifiers could consist of globally unique 32-bit integers that are mapped by means of hashing into a machine-relative identifier-to-address table—as in the V-System (see Chapter 7).

Given the ability to rebind a given reference, we must still be able to *find* all references to a program to rebind or we must retain rebinding information long enough to allow references to be rebound in a lazy evaluation manner. Systems that do not make use of broadcast communications must rely on techniques such as "back links" or forwarding addresses to do their rebinding. As mentioned in Chapter 2, these techniques are either slow and complicated or are fault intolerant. If accurate forwarding addresses must be maintained, then we must be able to garbage collect those of deceased programs in order to avoid overflows. One way to do this is for the local operating system of a machine keep track of whether the programs running under it were migrated to it and from where. When a migrated program dies, the operating system must send a message back to the original machine from which the program came telling its operating system that it can delete its forwarding address for the program. Since programs might migrate more than once, one can end up with chains of such "back pointers". To deal with the case when a chain is broken, we must rely either on broadcast to reconstruct the lost back pointer or on a fault-tolerant centralized server of some sort. (The latter approach will be discussed in the next section.)

On the other hand, if broadcast communication is available, then we can simply rebind references by having them broadcast to find the new location they are to refer to, as shown in Figure 4.5. If the kernel of the machine to which a program has migrated

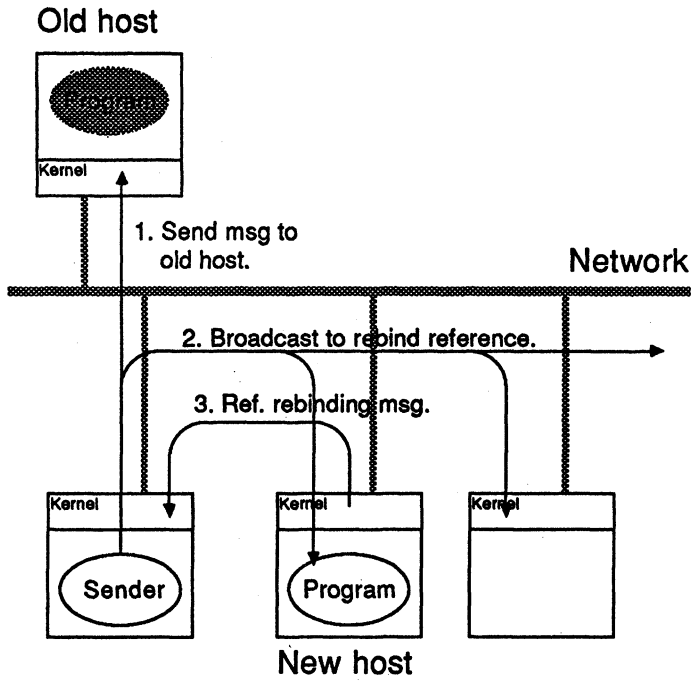

Figure 4.5: Rebinding references.

responds to such broadcasts with rebinding information, then only a single broadcast is incurred per reference rebinding. The overhead of rebinding can be reduced even further by including the rebinding information with acknowledgement messages that must be sent anyway and by rebinding *all* relevant references on a given machine that has received rebinding information. The latter is automatic if a machine-wide mapping table is used instead of storing the mapping information separately for each reference.

One disadvantage of our broadcast approach is that it requires that *all* references to locally non-existent processes—including dead processes—be retransmitted as broadcasts in order to ensure that the desired process hasn't migrated. As a result, every reference to a dead process will result in a needless broadcast and subsequent timeout delay in order to ensure that the dead process wasn't a migrated process instead. To avoid this problem, we can keep track of recently deceased programs in order to reduce the number of cases where broadcast retransmission must be employed.

If forwarding addresses are kept, then one might be tempted to consider keeping a reference count with each forwarding address in order to detect when a server has been migrated that has many clients whose references to it must be rebound. If a forwarding address is employed more than once, for example, a reference rebinding message would be broadcast to every machine in order to avoid having to lazily rebind every client's references to the server. However, this requires that *every* machine in the system must explicitly search its tables for possible references to rebind when such a broadcast rebinding message is sent. The overhead of this will almost always be greater than that incurred through lazy evaluation. Thus, the only advantage one might gain would be to reduce the load at the forwarding machine if having to rebind a great many references is considered to be too great a burden for a machine that is seeking to get rid of load.

Given the surprisingly small overhead incurred on the system by broadcast (see Chapter 5) and the complexity required to implement a forwarding scheme, we argue that simply

maintaining a list of recently deceased programs is by far the more preferable approach to choose. A relatively small table can substantially reduce the number of times that deceased programs must be detected by retransmission timeout. (See Chapter 5 for our assumptions about how often we expect programs to be invoked by the average user.)

4.3.5 Rebinding References Without Broadcast

If broadcast communication facilities are not available, then the reference rebinding technique described above won't work. Leaving a forwarding address on the original machine will work as long as that machine does not fail.[4] If we wish to ensure correct operation of the system in the face of failure, however, then we must provide a *reliable* place where forwarding addresses can be found.

This necessarily implies a well-known "forwarding" server that is *always* up or whose state can be quickly reconstructed after it has crashed. The latter would require every machine with forwarding addresses to monitor the server and then transmit copies of its forwarding information to the new instantiation when it appears. Stable storage doesn't help us since we must be able to get at the storage while the machine to which it is attached is down. The former approach requires replication. We will only address single machine failures; the extension to multiple failures is straight-forward. Replication simply requires that two well-known copies of the server exist that contain a copy of the forwarding information. The cost of updating two servers at migration time is small, so that this approach is simpler and cheaper than that of reconstructing the state of a single server.

Rebinding a reference to a migrated program now involves the following steps:

1. If the original machine is still up, then the reference can be immediately rebound.
2. If the original machine has crashed, then one of the copies of the forwarding server is queried for the information desired.
3. If the first copy of the server has also crashed, then we must query the second copy for the same information.

Note that we cannot simply *cache* forwarding addresses at the original machine since this would imply that determination of whether a referenced process is dead or has migrated would involve querying the forwarding server. In order to make this design work, we must garbage collect the forwarding addresses of deceased programs. However, unlike in the broadcast-based design, we cannot rely on broadcast to overcome back pointers that have been lost due to crashed machines. Instead, we must rely on the forwarding server to perform the necessary garbage collection. Thus, when a migrated program dies, the operating system of the associated machine simply sends a message to the forwarding server indicating this and lets the server contact all forwarding machines to garbage collect their copies of forwarding addresses.

In summary, one can get by without broadcast facilities to rebind references, but it requires the maintenance of replicated forwarding servers and makes migration slightly more expensive.

4.3.6 Effect of Demand-Paged Virtual Memory

If demand-paged (or demand-segmented) virtual memory is employed, then a different technique for copying address spaces can be employed because only part of a program's memory state will now be in main memory and the rest will be on the backing store used.

[4]Note that many failure recovery schemes, for example, the *Publishing* approach cited in [PM83], also rely on broadcast communications for their implementation. Hence, invoking failure recovery as a safeguard for forwarding addresses is questionable even if the other concerns cited in Chapter 2 were ignored.

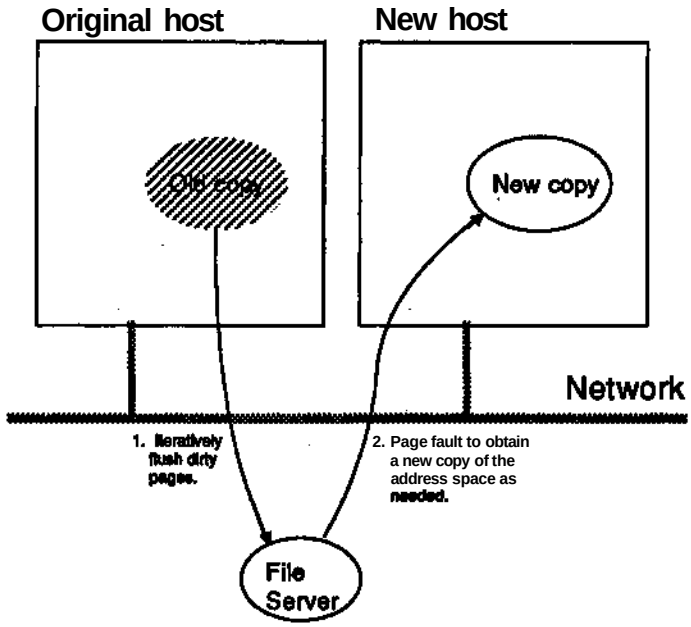

Figure 4.6: Migration with demand-paged virtual memory.

If we assume a global backing store, then copying address spaces can be done by flushing all of the program's dirty pages to the backing store and then simply paging to the new machine as needed. This is illustrated in Figure 4.6. Pre-copying would now consist of repeated flushing of dirty pages and the time required to pre-copy would depend on the page table sizes and normal flushing policy used by the operating system.

This approach to migration takes two network transfers instead of just one for pages that are dirty on the original host and then referenced on the new host. However, it allows us to move programs off the original host faster since we need not copy the entire address spaces at once, which is an important consideration. Also, the number of pages that require double copies typically should be small.

A very nice feature of this approach to copying state is that it requires no additional copying functionality to implement—we simply make use of the existing demand-paging facilities. An optimization one might consider to avoid the overhead of page-faulting the entire working set in to the new machine is to copy the final set of flushed dirty pages on the original machine directly to the new program instance. However, this would require extending the paging facilities of the system in exchange for a relatively minor decrease in migration start-up cost on the new machine.

If the backing store employed is not on a network storage server, but rather on the source machine, then we must also deal with migrating the backing store from the old machine's local storage server to the one on the new machine. This is a special case of the general problem of dealing with residual dependencies, which is discussed next.

4,4 Machine-relative Servers

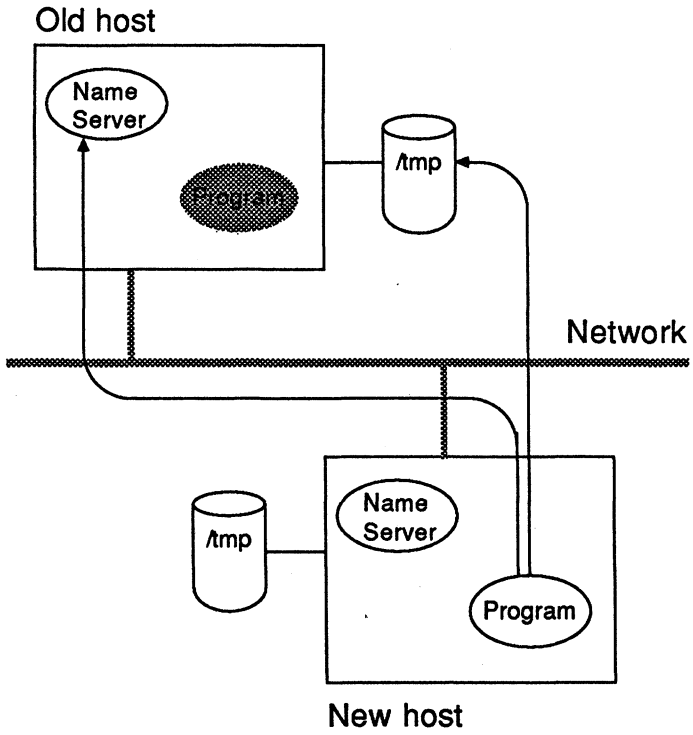

Figure 4.7: Residual dependencies on state in machine-relative servers.

The state of a program does not consist of just its address spaces and various descriptors for it in the kernel of the operating system. It also includes state stored in various local and global servers, as shown in Figure 4.7. As discussed earlier, when we migrate a program, we must migrate any associated state stored in machine-relative servers as well.

There are four issues that must be addressed in order to migrate the state of a program in a machine-relative server:

- We must copy the state information from the server on the original machine to the equivalent one on the new machine.
- If the server implements a local name space—for example, if a file server implements file names that are local in scope to itself—then the server must be able to unambiguously resolve any name conflicts that might occur when a program's state is migrated from one server to another. Sometimes this isn't possible, implying that the name space of the server must be made global in scope.
- All references in the migrating program to the server on the original machine must be rebound to point to the equivalent server on the new machine. Note that this is a different kind of reference rebinding than that needed for references to the migrating program itself. *References to machine-relative servers must be rebound to different objects, not just different locations for the same object.*

  Additionally, references to objects within the server by means of server-relative identifiers must be remapped so that they do not conflict with any mappings of server-relative identifiers that the server on the new machine has already established.
- Any operations requested of a machine-relative server that could modify the state of a frozen program must be deferred and forwarded to the equivalent server on the new machine.

In this section, we will examine each of the four issues in turn, using four common machine-relative servers that appear in many operating system designs. These are:

- The program manager introduced in Chapter 3, which controls the resources of the machine and implements the security and resource sharing policies desired by the machine's users.
- An exception server, which is responsible for bundling hardware exception interrupts/traps into a form that can be passed to local (and remote) exception handling processes. In many systems, this service is implemented directly by the operating system kernel.
- A local name server. As had been pointed out earlier, there are several different naming designs possible for an operating system. We will examine how the choice of design can make migration simpler or harder to deal with.
- A local file/storage server. We will treat local storage servers as a special case of local file servers since storage servers can be thought of as file servers that implement a flat name space of files, each of which represents a block of storage. Thus, for example, the problems mentioned earlier with local backing stores will not be directly addressed since they are a subset of those that must be addressed if the backing store were treated as just another file.

4.4.1 Copying State Information

There are essentially two ways that one can go about copying the state information of a machine-relative server:

- The server can implement query and install operations, so that a third party can extract the information to copy from the original server and can then install that information into the new server.
- The original server can directly copy the information to its counterpart on the new machine.

The advantage of the first method is that the server frequently needs know nothing about migration—it merely provides query and install operations that are of general use. (How much a server needs to know about migration will be discussed as part of deferring requested operations.) The disadvantage is that this method is slower than the second. If the state information in the server is stored in a simple linear form, then query/install will only take twice as long as a direct copy would. But, in the more likely case that the state information is stored in a complex set of data structures, the time required to "unpack" and "pack" the information for the associated query and install operations can make migration take considerably longer than if the server can transfer the information to its counterpart in an encoded manner that both understand.

If the amount of state information to be copied in a server is relatively small, then the overhead of using the slower query/install approach may not be significant compared to the overhead of other aspects of migration. In such cases, the benefit of not having to force servers to understand about migration outweighs all other considerations. Of our four example servers, three fall into this category, namely, the program manager, exception server, and name server. Each of these stores program state information that is typically less than a kilobyte in size. The times required to query these servers for a program's state and then reinstall it is typically on the order of milliseconds to tens of milliseconds, which is still within the range of acceptable freeze times as long as there aren't more than a few servers to deal with.

The example of a server for which this approach is not acceptable is a local file server, which may contain files whose sizes are measured in megabytes and whose copy times would

be measured in seconds. In order to achieve acceptably small freeze times for migration, we must not only require that the server implement direct copy of its state information, but also that it employ pre-copy techniques in order to be able to do most of its copying outside the time that the migrating program is frozen. Thus, a local file server will not only have to understand about migrating its state information, it will also have to understand "pre-copy" requests that tell it when to perform the next pre-copy iteration, which must be coordinated with the pre-copying of the program's address spaces. Furthermore, it will have to implement "dirty bits" for its files in order to implement pre-copying at all. Consequently, part of the state information for an open file will have to be a bitmap of some sort that indicates which blocks of the file have been written since the last time the bitmap was cleared. While all these requirements for the file server are feasible, they almost certainly represent a *major* change to its implementation.

4.4.2 Name Conflicts

In order to migrate the server state of a program, we must be able to both find it and to install it in the new server without causing any conflicts with names that already exist in the new server. This requires that servers implement either a global name space or provide a name context for each program (since that is the unit of migration). They *cannot* just use a naming context that is local to each server. Note that we are talking about high-level server-relative names, such as file names and directory names (e.g., `/tmp/foo`), that have no unique meaning in a global sense; i.e., their global interpretation is as a reference to a *set* of objects, of which there may be at most one in any given server. Section 4.4.3 discusses the implications of migration for globally unique object identifiers.

The canonical examples of servers that must deal with this problem are local name servers and local file servers. Each will be used here to illustrate different aspects of how one can deal with naming conflicts.

If an operating system's name service is structured appropriately, then it will cause little or no problem. For example, Chapters 3 and 7 describe a design by Mann and Cheriton that obviates the need for a machine-relative name server by pushing naming into program-local library routines and the servers that implement the named objects themselves. Another way to avoid the issue of migrating name service information is to make machine-relative name servers global in the sense that the server on the user's machine is used to provide name service for all the user's programs, wherever they may be. However, this implies that all other machine-relative servers must be globally known to the extent that the name server on a user's machine must be able to refer to the machine-relative servers of a remotely executing program's machine. One might also argue that this second approach will suffer from performance problems since name service is now provided remotely, even for local servers. However, if the name server is used only for the initial access to an object, then this should not be an issue. Finally, the least desirable design choice, from the point of view of migration, is one in which name service is provided by the machine-relative name server on the machine where a program is running. In this case, we must implement the full migration procedure, including copying name binding state and storing it such that programs' name bindings won't conflict with each other. Note that we already faced similar problems when providing simple remote execution.

The potential name conflicts that may occur with file servers cannot all be solved since objects can be created that outlive their creators. In order to allow `/tmp/foo` to refer to two different files, depending on which program is using the name, names must be resolved relative to the program using them. But once a program exits, the files it has created no longer have unambiguous names since the context in which their names are to be interpreted no longer exists. The only way that this problem can be remedied is to make the name spaces of the system's local file servers global in scope.

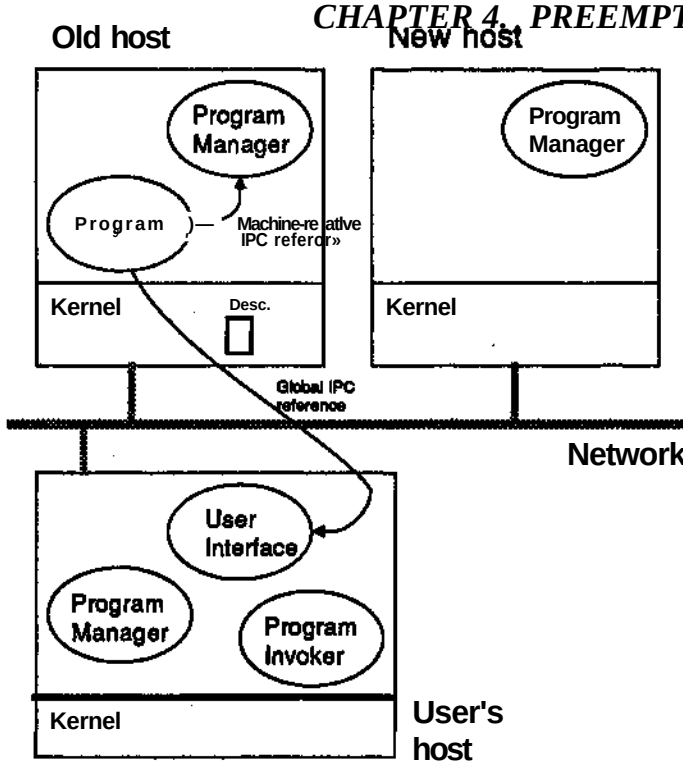

Figure 4.8: Global and machine-relative IPC references.

4*4,3 Rebinding References

There are two aspects to rebinding references to a machine-relative server:

- References to the server itself must be rebound to the equivalent server on the new machine. Figures 4.8 and 4.9 illustrate how such references differ from "normal" IPC references.
- Migration implies that object identifiers that are intended to be unique relative to a server might now point to more than one object within the same server. This implies that such identifiers must be interpreted relative to some context in order to achieve a unique mapping to the objects they are intended to designate.

Being able to rebind the same reference to different servers requires that the reference become *well-known* to the operating system in essentially the same way that was described in Section 3.2 for allowing references to machine-relative services independent of the location of a program. The level at which interpretation of well-known names is performed at will determine how much overhead is incurred with each use. For example, if the system's name service facilities are used, then *all* references to a machine-relative server must pass through the name service facilities. If we wish to avoid most of this overhead, then the system's design must be modified to understand that some subset of its lowest level name space, namely the space of globally unique identifiers used for IPC, will consist of well-known identifiers that must be mapped in a context-dependent manner. Note also that the interpretation of well-known names or identifiers must be implemented on a global scale since references to the machine-relative servers of a program may exist on machines other than the one the program is running on.

Exactly how a system implements well-known references at the IPC level is relatively unimportant. The crucial point is that the system's IPC facilities must now not only

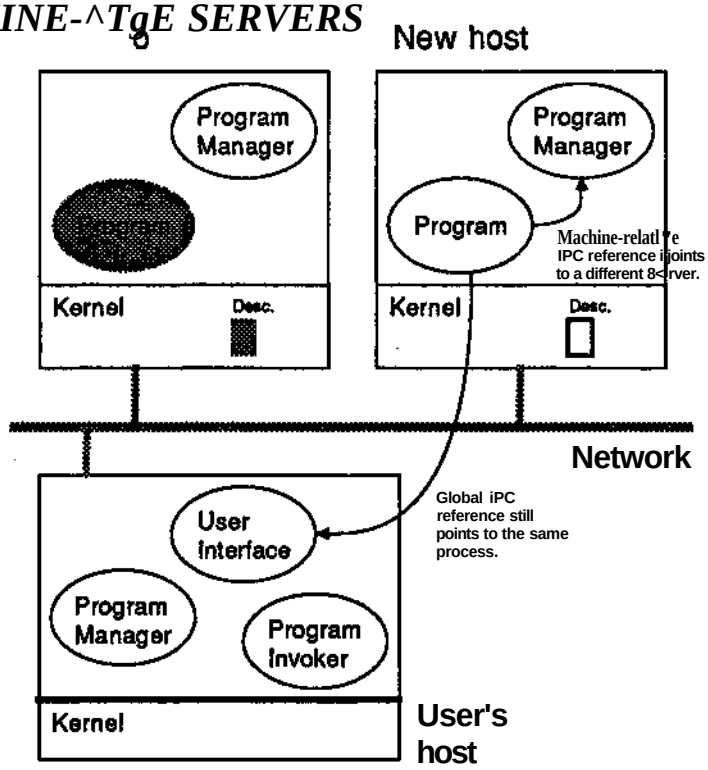

Figure 4.9: Rebinding a machine-relative IPC reference to a *different* server.

provide one level of indirection so that references to the same object can be rebound when it moves, they must also have an efficient means of determining whether a given reference refers to the subspace of names that must be specially interpreted. This can be done, for example, by having pre-defined port indices in systems such as Demos/MP and Accent, or by reserving one bit of an identifier's representation to signal a well-known one, as is done in the V-System (see Chapter 7).

The second aspect of rebinding references, namely, relative scoping for the local identifiers of a server, implies that objects can no longer be unambiguously referred to by means of a "context pair" consisting of a server's globally unique identifier plus the unique identifier of the object relative to the specified server. Instead, a somewhat different context pair can be used. This new context pair consists of the well-known reference for a server relative to a particular program and the unique identifier of the object relative to this well-known reference. That is, the server part of an object's context pair encodes the program that the object is associated with. Note that when a server encounters an object identifier in **a client request message, *the server must be able to determine what well-known reference was used to reach it*** in order to determine which context to interpret the identifier in.

4.4.4 Deferred Server Operations

The final issue is deferral of requests made to a server that can modify the state of a frozen program. Note that query operations can still be performed without delay. However, modify operations must be forwarded to the equivalent server on the new machine for application after the frozen program has been completely migrated.

The principal problem is one closely related with how references to servers axe rebound: If operations are not all idempotent, then we must ensure that the original and the new

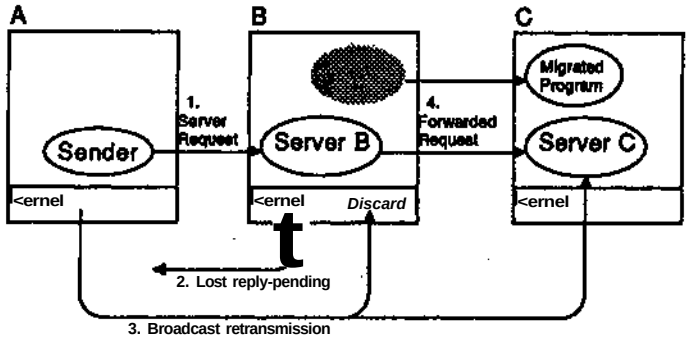

Figure 4.10: Generation of a duplicate server request message.

server do not mistakenly both receive the same request message and end up executing the requested operation twice. This can happen if, for example, the normal IPC reference rebinding scheme broadcasts a request message because of a lost "reply-pending" packet during a period when both the original and the new server are able to receive messages for the associated program. Figure 4.10 illustrates the problem. Server B has received the request (step 1) and will eventually forward it to Server C (step 4). The kernel on machine B sends a normal "reply-pending" packet for the deferred operation to machine A that gets lost (step 2). As a result, machine A retransmits the request as a broadcast (step 3). Machine B's kernel discards the message as part of normal duplicate suppression, but machine C's kernel has never seen this message before and delivers it to server C. Unless the forwarded request of step 4 is detected as a duplicate, we end up with server C executing the request twice.

If a server can be structured to have all operations that involve migratable state be idempotent, then the question of duplicate suppression can simply be ignored. If the kernel is able to uniquely identify request messages; for example, if the kernel can detect forwarded request messages as being duplicates of request messages already delivered, then we can rely on the IPC facilities to detect duplicates. Otherwise, all "non-idempotent" servers will have to implement the necessary duplicate suppression themselves.

This is not a trivial thing to provide: The server must now maintain state for all its clients so that the new machine's server can detect when a request is forwarded to it that has already been received locally. That is, machine-relative servers can no longer be connectionless in nature.

4.4.5 Avoiding the Problem

A much simpler approach to dealing with machine-relative servers is to avoid their existence and use where possible. We have already outlined how name service can be provided without the need for a local name server. If machines are diskless and employ network file servers for their file access, then we can avoid the existence of local file servers altogether. Several studies have shown that the performance obtained from systems that rely strictly on network file servers can be comparable to, if not better than that obtained when file service is provided from local file servers on each machine[LZCZ84]. In essence, the more

location-independent the design of the system becomes, the easier the task of migration becomes.

Another way in which the complexity of dealing with machine-relative servers can be reduced is to avoid the proliferation of specialized servers where it is reasonable to do so. For example, there is no real reason why the program manager and exception server cannot be merged into a single server, since both manage different parts of the state information that determines how a program is to interact with the basic "virtual machine", namely the hardware and local operating system. Similarly, providing separate storage and directory servers instead of a combined file server would be disadvantageous from the point-of-view of migration concerns.

4.5 Failure Considerations

Migration may fail either because a machine can't be found that is able and willing to accept the migrating program or because the machine being migrated to fails during the migration procedure. What to do with a preempted program that can't be migrated because no alternative machine can be found is a policy issue that we will not address. However, when the migration procedure itself is interrupted by machine failure, we must be careful to leave the original instance of the program in a consistent state before proceeding any further. If the program has already been frozen, then we must unfreeze it to avoid inadvertently causing timeouts through the additional delays that would be incurred if we directly attempted a second migration. These considerations imply that we should simply abort the migration procedure and start over from the beginning.

The hardest aspect of aborting migration involves correct recovery of deferred interactions with the migrating program. For the case of deferred kernel operations, if they are the responsibility of the invoker, then the invoker must be made to realize that they should now be re-executed rather than forwarded. If the kernel queues them, then it must be careful not to forward them until the point at which the migration is irrevocably committed to finish. Simply unfreezing the program can then be used as the signal to execute all deferred kernel operations.

The same retention requirement also applies to all messages queued. In light of this requirement, the most appropriate point at which to forward deferred kernel operations and queued messages is when the old program instance is destroyed. Since reference rebinding occurs after this point (namely, once the new program instance has been unfrozen), this will maximize the time during which the migration procedure can be aborted since all other aspects of the procedure are idempotent.

Machine-relative servers face the equivalent problem and must similarly take care to forward deferred operations as late as possible and be prepared to abort their migration procedures (e.g., pre-copying) beforehand.

4.6 Security Considerations

Migration of a program implies the authority to do so. More specifically, it implies that the entity implementing migration of a program—the "migration manager"—must have the requisite access and control privileges to the state of the program. It[5] must have transparent read access to the address spaces of the program in order to pre-copy them, it must have the ability to suspend execution of the program, and it must be able to obtain a linearized representation of the program's kernel state for passage to the kernel of the machine being migrated to (assuming that installation of kernel state can only be done by

[5]The migration manager may be either a program or a server. Making it a program would imply that the implementation code and data need only be present when needed.

local invokers). Finally, it must have the necessary authorization to be able to implement the migration of state in machine-relative servers. Unless the migration manager has blanket "system" privileges, this last prerequisite may require access authorization that is not kept by the kernel of the operating system. Program debuggers require a similar level of access privileges.

The decision that must be made is what kind of access privileges we are willing to confer to a migration manager. The simplest approach would be to give it full "system" privileges so that the kernel and all servers will allow it to do as it pleases. But this is generally considered to be a mistake since it provides the opportunity for an erroneous program to wreak havoc. The alternative is to limit access privileges in some manner.

Most systems support the concept of an owner of an object and confer various privileges to the owners of objects. One might consider attaching the privileges we need for migration to ownership and then forcing ownership privileges to be passed to the migration manager when migration is invoked. However, unless ownership privileges can be conferred to multiple entities simultaneously, we will run into the problem that conferring ownership privileges to the migration manager precludes the previous owner from privileges it thought it had and may need for correct operation. For example, we cannot migrate a program that is also being debugged. Thus, ownership must be more of a capability (or an equivalent access list) that can be handed to multiple *processing* agents than something that is associated with a single active entity. For example, associating ownership privileges with a user account, so that any process that runs under that account can obtain them, would provide the necessary functionality. In contrast, associating ownership privileges with individual processes will clearly not provided the needed functionality.

Note that system designs that rely on the concept of a unique owner to implement various operations (for example, notification of the death of a process) would have to be changed to distinguish this form of ownership from the kind of "ownership privileges" we have been talking about. In general, it may be simplest to just go to a general access list or capability-based scheme in which ownership is not the sole key to access privileges.

4.7 Summary and Conclusions

In this chapter, we have dealt with issues of how to preempt a program from one machine by transparently migrating it to another machine during execution. Although one could preempt programs simply by destroying them, this loses work that has already been done and may alter the failure characteristics of the system such that the probability of multiple components of a (fault-tolerant) application failing within a short period of time is significantly higher than before.

The two principal issues one must address with program migration are those of avoiding excessive interference with the normal functioning of the system and dealing with the residual dependencies that are caused by programs having state in machine-relative servers. Be introducing a technique we call *pre-copying*, we have shown that interference with the execution of the migrating program and the system as a whole can be reduced to a level the system must already be able to handle as a result of other types of interference. In particular, critical system servers, such as file servers, are not subjected to inordinate delays when communicating with a migrating program. Pre-copying works by suspending program execution and operations on the program by other processes only for the last portion of the copying of the program's state, rather than for the entire copy time. As a result, the freeze time of a migrating program is reduced to an interval that is similar in length to the delays caused by operating system operations such as swapping a program to secondary storage in a paged virtual memory system. Note, however, that pre-copying does not reduce the time needed to migrate a program from a machine; i.e. the time interval between when a program is first preempted and when it is actually gone from a

machine is not reduced by pre-copying. In Chapter 7, we present measurements that show that the total time needed to migrate programs less than a megabyte in size is on the order of a few seconds if 1 Mips processors and a multi-megabit network are used as the hardware base.

In order to limit the residual dependencies a migrated program has on the original machine it was executing on, one must do one or both of two things:

- Migrate the state that is associated with a program in each machine-relative server and rebind references to it on the original machine to refer to the equivalent server on the new machine.
- Eliminate the machine-relative server from the design of the system.

If the state of a program in a machine-relative server is to be migrated, then the design of the server must meet various constrains, which we have discussed in Section 4.4. Meeting these constraints may require the server to be designed in a manner that is fundamentally different from that which would be chosen if migration did not need to be supported. For example, it may require a different formulation of the object naming semantics provided, and may require significant differences in implementation to support pre-copying operations. The canonical example of a server that might have naming problems and that would have to support pre-copying operations because of the large amount of state it stores is a local file/storage server.

We have also shown that migrating state between machine-relative servers imposes a new requirement on the communications facilities of the operating system. Rebinding references to such servers requires that the same identifier refer to a *different* server and not just a different location for the same server. Unless server references are reinterpreted on every use, this requires that machine-relative servers be referenced by means of *well-known* server-ids since the operating system must treat them differently from other server-ids.

A different approach to the problem of residual dependencies, which we advocate, is to avoid the use of machine-relative servers where possible. For example, name bindings can be stored in a cache in programs' address spaces as well as in the servers that implement named objects instead of in machine-relative name servers. More importantly, if file/storage service is provided by network file servers, then we can avoid having to deal with the one server that causes by far the most difficulty. Note that we are *not* advocating the collapse of operating system services into a monolithic kernel, as is done in UNIX. Rather, we are advocating increased use of global servers and placement of state information into the address spaces of the programs themselves. In general, the vast difference in size and type of state information stored by various machine-relative servers implies that one must deal with each potentially machine-relative service individually in order to obtain a suitable design that avoids unnecessary complexity and overhead, but still achieves the overall goals required of the system.

Another point of view on the issue of machine-relative servers is that migration facilities are more or less difficult to provide—hence, more or less suitable—for systems depending on the architecture a system imposes. For example, if every machine has a local disk, then it is may be that machine-relative file servers will be strongly desired. Similarly, if multicast is unavailable, then it may be difficult to avoid having machine-relative name servers. From the point of view of providing migration facilities, the most suitable system architecture is one in which non-server machines are all diskless and the communications medium is capable of efficiently supporting multicast. The least suitable architecture would be the converse one in which every machine has a local disk and is connected to a network that supports only point-to-point communications.

The final issue, which we only briefly examined, is that of security and the access-control mechanisms needed to implement and control the use of migration facilities in a system. In order to gain access to the state of a migrating program, the operating system must confer the necessary privileges to the entity implementing the migration procedure, namely the

"migration manager". This entity may be an agent of the local user of a machine that wishes to migrate someone else's programs or it may be an agent of the (possibly remote) owner of a program. Since migration may take place concurrently to other activities that require similar access privileges, such as debugging of a program, these access privileges must be conferrable upon multiple processing entities. That is, a design that provides such privileges to only one processing agent, will not be able to support transparent migration concurrent to other activities such as debugging.

In addition to dealing with the major architectural issues just presented, we have also addressed several implementation issues that must be dealt with in order to correctly provide transparent program migration. The two most significant results are:

- In order to defer receipt of messages sent to a frozen program, they must be buffered somewhere and communications timeouts that would terminate a message transaction must be prevented from occurring during the deferral period. One can rely on the system's retransmission facilities to buffer reliably delivered messages. Simply dropping unreliable messages is not acceptable since it changes the failure characteristics of the system in a way that may preclude higher layers from correctly implementing communications in the face of migration. In general, we face the problem that higher levels of the system may not be aware of the delays that migration can cause and must, thus, rely on the fact that pre-copying prevents the freeze time of programs from becoming too long.
- In order to transparently rebind references to migrated programs, the operating system must be able to either find all IPC identifiers before they are used or detect them when they are being used. Unless location addresses that a program has stored at random locations in its state information can be found by the operating system in order to be rebound, IPC identifiers must contain at least one level of indirection before being bound to a particular location address. With broadcast communication facilities, references can be rebound very simply by broadcasting them after the first or second retransmission of a message fails to find its intended destination. However, one should prevent a situation where messages to deceased processes must rely on broadcast retransmissions in order to determine that the processes haven't migrated instead.

Chapter 5

Finding Idle Machines

5.1 Introduction

In this chapter we will address the issue of finding an idle machine for remote execution or migration without considering questions of fairness and load balancing per se. The goal here is to find the "least loaded host" subject to cost constraints. As described in Chapter 2, others have already shown that without knowledge about the nature of individual programs, such as expected completion times, the best load sharing algorithms to use are variations on placing programs on the least loaded machine. "Least loaded" is determined by a metric such as job queue length or processor utilization. We will not consider algorithms that make use of program-specific information since, in general, such information is not available. The principal cost constraint we will be concerned with is the cost of gathering state information from the system rather than the cost of performing inferences from such information, as is the concern in, for example, [RS83].

As pointed out in Chapter 2, our environment is not suitable for scheduling designs based on locality of information or fixed querying schemes. Machines may be surrounded by "rings" of unavailable machines and membership in the set of available machines for remote execution or migration is dynamic, with a large number of possible candidates. Therefore, we must either obtain state information from *every* member of the system or must exclude members by dynamic criteria such as load rather than geography. Since scalability is an important consideration for us, the latter approach of finding methods for excluding machines from the set whose state information we must gather will be an important aspect of our work. It will be shown that substantial reductions can be made through appropriate use of multicast and that accurate state information frequently need be obtained from only a few hosts.

The principal thrust of this chapter is to demonstrate that facilities for finding suitable hosts for remote execution or migration can be constructed that perform well, are highly available, and can be scaled to systems containing hundreds of machines. Assuming the availability of efficient broadcast facilities, we will further show that both centralized and decentralized designs can be constructed that are similar in their performance and fault tolerance characteristics. Whereas our decentralized designs will appear scalable to hundreds, or perhaps even a thousand machines, our centralized design will be scalable to several thousand machines. However, in exchange for less scalability and perhaps somewhat less flexibility in dealing with issues discussed in Chapter 6, a decentralized approach can offer greater simplicity of design.

An important point to keep in mind is that our approach will be to seek an overall understanding of the suitability of the designs and facilities examined rather than to attempt a detailed analysis of their specific performance characteristics. This approach is taken for several reasons:

- The workload assumptions we must base our analyses on are conjectural at best, especially for those situations where a detailed analysis would be most appropriate.
- The designs will be shown to work well enough and be scalable to system sizes sufficiently large that we will argue that detailed analyses cannot significantly affect our results. That is, the important parameters of the designs can be determined using relatively simple calculations. Furthermore, we will argue that factors external to scheduling, such as network bandwidth, should dominate the scalability of our design parameters far more than the scheduling considerations we will concern ourselves with here.

5.2 Assumptions

In this section, we describe in detail the assumptions we make about environment and workload. To begin with, we assume that each machine's local operating system employs preemptive scheduling, possibly with multiple priority classes. While we assume autonomy of machines with respect to obeying external scheduling decisions, we also assume that machines do not lie and are willing to participate in global information gathering schemes. Thus, all machines are assumed to obey the global scheduling protocols we will devise for updating state information or answering information queries.

5.2.1 Workload Assumptions

As discussed in Chapter 1, our principal interest is an environment in which most machines are personal workstations that are dedicated principally to serving their local users. However, while we assume that most machines will have logged-in users, we also assume that there are many essentially idle machines available. "Idle" is defined here to mean local activity of less than a few percent of a machine's resources, averaged over the period of a typical remote program execution, which we assume to be on the order of one to a few minutes. Aside from the empirical evidence obtained from an actual implementation, we point out that this is reasonable when one considers that the dominant activity performed by human computer users is editing—of program files, or text manuscripts, or illustrations. On contemporary workstations, editing can be handled with sufficiently few resources to qualify the workstations as being idle by this definition.

In this environment, while "active" local users will create workloads on their machines consisting of interactive applications that typically exhibit a variable and bursty load behavior, many machines will have a fairly stable load behavior. More importantly, exactly those machines of interest to our schedulers will have a stable load behavior that is basically the same across many machines, namely "*idle*". However, given the unpredictability of local activities, the best a scheduler can do is to *predict* the near-future activity of any machine by extrapolating its recent past activity. Although one can envision machines keeping track of or knowing the usage patterns of their local users, we will assume that the only information available is the load history over the recent past. This can be used with such traditional prediction models as extrapolating the current load to a future time equal to the time already spent at that load. Thus, the longer a machine is idle, the more likely it is assumed to stay idle.

One area frequently mentioned when dealing with scheduling of programs is use of estimated program resource requirements to make scheduling decisions. Unfortunately, many of the programs we assume will be run by most users don't know much about their resource requirements other than the minimum amount of memory required to load the program. Frequently, their processing and memory requirements are a function of the input given them and, hence, would require that someone calculate them before each program invocation. While this may be reasonable for large long-running programs, we claim that

users will be unwilling to perform such calculations for every program compilation and text formatting job they run. Therefore, we will ignore this aspect of scheduling, treating it as a potential optimization for future consideration.

A variation that we must deal with is long-term cycles such as day/night cycles and events such as meetings and lunch. These imply that the system's workload is inhomogeneous in nature, by which we mean that the time-averaged number of remote execution and migration requests generated by each host in the system are non-constant functions of both time and host-id. Whereas during the day most machines will have local users, which might be assumed to each generate roughly equal numbers of programs to remotely execute, during the night there will be far fewer users and there may be considerably more long-term applications or distributed applications that generate many programs from a single source point. Similarly, transient events, such as lunch or meetings, can create sudden changes in the overall system load or pockets of available resources. Thus, we are faced with designing scheduling facilities that must work well under a variety of very different workloads.

Such variations in workload bring up the question of how the generation of programs for remote execution and migration is distributed. We will assume that the distribution between users is random and can be satisfactorily modeled by a Poisson distribution. Distributed applications will, however, generate correlated sets of remote execution requests. It is difficult to characterize the distribution of such requests since the time interval between requests within an application is a function of the application itself.

We will assume that the average number of requests per application is small, say, between one and five. We argue that this is reasonable for the current state of computing and that, in general, most applications, such as document preparation or compilation, will not generate a great many subprograms. This is based on a locality argument that people do not frequently change a great number of different files between, for example, recompiling a program or reformatting a text document. We, thus^ assume that *large* distributed applications occur infrequently or at unusual times such as during the night when there are few users.

We will also claim that host selection requests from distributed applications will almost always either be batched into one "multiple" selection action or will have program loads interspersed between them. Since program loads typically take several seconds to perform and different programs will take different amounts of time to perform, we argue that in the latter case there will effectively be no correlation between host selection requests. We base our argument on the following reasoning: If a program wishes to start several distributed subprograms immediately and the interface for requesting host selections allows specification of multiple selections just as easily as a single selection, then the program's implementor will almost certainly go ahead and request all the host selections at once. If the program does not immediately know all the subprograms it wishes to invoke, then it will very likely perform each program load immediately after having selected a remote host for it because that will allow it to "finish" the task of determining the current subprogram to execute so that it can go on to determine the next subprogram to execute without having to worry about unfinished business.[1]

An important question for scalability is that of how many host selection requests are generated by each user (i.e., each machine) per unit time, *k.* We will assume that *active* users, on average, generate at most one application per minute. This corresponds to informal observations we have made that most users spend *at least* one minute thinking, debugging, or text editing, between submitting tasks such as compilation or text formatting for execution. While we claim that most users are actually idle by our definition of the term, we will assume that all users generate one application for remote execution per minute all

[1] **Note that the former approach corresponds to behavior one might expect from a distributed application whose decomposition into subcomponents is determined by the application's designer whereas the latter approach corresponds to behavior frequently exhibited by automatic decomposition tools such as, for example, parallel "make" programs.**

of the time in order to obtain a "bounding" estimate. Since our entire derivation of host selection request frequency must rely on human factors that may change in the future with changing software tools, our assumptions can serve at best as crude estimates. We have, therefore, chosen a host selection frequency that is unnecessarily high in comparison to empirically observed values in order to examine our designs' performance under pessimistic assumptions. Note that if the average number of subprograms per application is d, then our assumptions imply that d remote execution requests will be generated per machine per minute.

We also assume that since remotely executing programs run for periods of time on the order of minutes, times for finding a remote machine of several seconds will be acceptable. This assumption turns out to be useful, not so much for dealing with the average response time of host selection requests, as for allowing a design to incur occasional delays in response time under exceptional conditions.

5.2.2 Evaluation Metrics

Most scheduling algorithms depend on a scalar metric for the load on a given machine. The most common quantity used is job queue length. However, since we assume that interactive applications are common, job queue length (or number of programs executing in our case) does not accurately reflect the fact that the load presented by any given program may be very variable in nature. For example, a workstation may have an interactive CAD application, a text editor, and a mail program all running at the same time. Such a workstation should not be viewed as having a load of three. Processor utilization is a more suitable load metric for our environment since it represents the actual number of CPU cycles being used, averaged over some time interval. However, if the time interval over which averaging is done is too long and there are few "idle" programs in the system, then job queue length may actually yield slightly better results because it can more accurately determine when programs terminate and, hence, when a machine becomes less loaded as far as future host selection actions are concerned. Chapter 6 discusses this topic and the question of how choice of load metric can influence issues of fairness in more detail. For purposes of this chapter, the choice of load metric is not important as long as each machine can measure it and there exists a simple procedure for comparing two machines' loads.[2] We will assume a processor utilization-based load metric for the remainder of the chapter.

How well a set of global scheduling facilities work can be measured in a variety of different ways, depending on what characteristics are deemed most important. Since autonomy is assumed from the beginning, we cannot accept too high an overhead incurred at machines that are not making use of the scheduling facilities. We will arbitrarily consider 1% of a machine's resources to be an acceptable upper bound on such overhead.

Since remote execution and migration cannot function without some means of finding suitable remote machines, we view availability as our second most important consideration. As mentioned earlier, since our goal for remote execution is to distribute programs that typically run for times on the order of minutes, we consider selection response times up to a few seconds in length to be acceptable and will emphasize availability over shorter response times. For example, occasional response delays of several seconds to allow reconfiguration of the scheduling facilities will be viewed as acceptable.[3] Our goal for average response time will be less than one second.

---

[2]Obviously, lower bounds on fixed quantities such as non-demand-paged virtual memory must be taken into account in any scheduling decisions made. One might also consider using a vector load metric, such as a triple indicating processor utilization, memory utilization, and network interface utilization. On machines with local disks the metric might include the disk utilization.

[3]One might argue that response times of tens of seconds are justifiable if one still achieves an overall gain in *user* performance. However, human factors considerations make such long delays undesirable. Fortunately, we will not have to consider such response times anyway.

5.2.3 Hardware Assumptions

As discussed in Chapter 1, we assume a communications medium with speed and failure characteristics like those of Ethernet[4] or token rings and that can provide efficient broadcast or multicast communications between machines. Although we will briefly discuss how our centralized design could be made to work without the availability of an efficient broadcast communication medium, such facilities will be used as a basis for most of the work to be described.

An assumption that critically affects our designs is what kind of network interface hardware is available on each machine. If multicast communication is used to send a query to many receivers, many of whom will return a reply message, then the network interface of the querying machine can be swamped with more messages than it can handle. Depending on the buffering capacity of the interface, and also of the rest of the machine and software, a significant fraction of the returned replies may be lost. In large systems of machines with buffering capacities of only one or two messages, the loss rate can easily exceed 50%[CZ85]. It is also important to realize that designs must be targeted for the *poorest* interface and smallest buffering capacities in use in a system, even if only a few machines have them. We will assume that network interfaces are capable of receiving every second network packet of a stream of "back-to-back" packets and that machines and their operating systems are capable of buffering at least 20 to 40 packets before reaching the limits of their capacity. This corresponds to our empirical observations on the performance of a SUN-2 workstation with a 3COM interface board connected to a 10 Mb Ethernet. The main point is that we assume that significant losses can occur if reply messages are not somehow spaced slightly apart and that machines are capable of buffering at least some small number of messages from a "reply set", but not necessarily more than a small number.

Closely related to the question of network interfaces is that of memory capacity and processing power. Our target environment is one containing machines with processors that range in speed from 0.5 to 2 Mips, and that have 1 to 4 Mbytes of main memory. Use of processors more powerful than 2 Mips will not significantly affect our designs in that factors other than processor speed will predominate. Note that 0.5 Mips processors already include "lower end" machines such as IBM PC/AT'S.

5.2.4 System Sizes

An important question concerns the number of machines in a distributed system. In small systems, almost any means of gathering state information will usually work well. Our goal is to support systems containing hundreds of machines. We consider a system of 100 to 200 machines to be the "minimum" size that must be supported and will aim for designs that can scale to between 500 and 1000 machines. Since our work is targeted at domains whose extent is limited by a single network, one can argue that larger sizes can be excluded for other reasons. Extensions into a (larger) internet environment will require changes to our designs anyway (see Chapter 8).

5.3 Centralized Scheduling

In this section we discuss centralized scheduling designs. By centralized we mean that state information is collected at a single logical location at which all scheduling decisions are made. In contrast, we define (completely) decentralized scheduling to mean that every host in the system performs its own scheduling actions for operations such as host selection. Note that if broadcast facilities were not available, one would have to choose a centralized

[4]Ethernet is a trademark of Xerox Corporation.

design since propagation of state information to every host in the system by point-to-point communications would be prohibitively expensive.

5.3.1 The Basic Model

Our basic model of centralized scheduling assumes "physical centralization" in additional to "logical centralization" so that issues of replication need not be addressed. Replication for performance reasons will not be necessary and Sections 5.3.4 and 5.3.5 will discuss issues of replication for fault tolerance.

Collection of global state information in a single location is best achieved by having machines send update messages containing their current state to a central server in an independent manner. One might consider having the server *query* the system for state if broadcast facilities are available. This would take care of stale information problems, but the same query could just as well be performed directly by clients, resulting in a decentralized scheduling scheme. Having a central server perform the state queries would reduce message traffic if state information was used to answer more than one host selection request, but this would then cancel the benefits gained with respect to stale state information. Thus, the basic model of a central server should consist of machines periodically sending update messages and clients sending host selection requests. Since machines can change their load at any time due to local activities, the scheduling server cannot rely on its selection history to infer the current system load.

The most important quantities that determine a scheduling server's performance are the frequency of selection requests, the frequency of state updates, and the service times needed to process a selection or update request. We have already stated our assumptions about selection request frequencies. For idle machines, we claim that state updates will occur only infrequently. For active machines, we need to filter out transient load changes by using time-averaged load values. To notice transitions, we want relatively short averaging periods in order to get frequent updates. To get upper bounds on server performance, we should bias our assumptions toward high update frequencies. We argue that once every 10 seconds is the highest frequency one should consider, given our assumptions of remote program durations of a minute or more and our desire to filter out insignificant local events (e.g., a brief flurry of editor page scrolling events). In practice, many machines will have much lower update frequencies since they should only send updates if a significant load *change* occurs, for example, a change in processor utilization from 10% to 50%.

We will now present the equations we use to estimate the performance of a central scheduling server. As mentioned already, the principal parameters of interest are:

- JV, the number of hosts in the system.
- 5_r, the time required by the server to process each host selection message. We assume that the server maintains an ordered list of host candidates so that the time required to select the best choice, subject to constraints on resources such as memory and on machine type, is essentially a constant.[5]
- JVp, the number of host candidates maintained.
- s_u, the average time required by the server to process each host status update message. If we assume that update messages occur at random from various hosts, then the time required to insert a host's update information into the ordered list of host candidates should be either linearly proportional to N_{gj} or constant, depending on whether or not the updating host becomes a member of the candidate set.[6]

[5] **Most idle machines should usually be able to satisfy typical resource constraints so that we expect that the first entry for any acceptable machine type should typically be chosen.**

[6] **We assume that if the host was previously a member of the candidate set, that it can be removed from the set in constant time; for example, by maintaining a hash table keyed on host id for the candidate set.**

- it, the average number of applications generated by each user (i.e., each host) per unit time.
- cf, the average number of programs that a remotely executing application consists of. Thus, the average number of host selection requests generated per host is $k \cdot d$.
- *fi* and /u, the frequencies of periodic updates assuming the worst case of continual change. Since idle machines should generate substantially fewer updates than machines with active local users, we employ two update frequencies to reflect these differences.
- t_{msg}, the overhead of a message transaction with the server.

Given these definitions, the number of host selection requests the server must process per unit time is:

and the load on the server due to these is:

$$l_{\overline{r}}:N'S\sim_{t}N^{*}k_{r}^{*}d^{\mu}\$ \qquad (5.2)$$

To obtain an estimate for update message load, we will assume that all hosts in the system generate periodic update messages with a frequency of either *fi* or f_u. If we assume that only the N_g most idle machines generate updates at the lower frequency of /t-, then the load on the server due to update message traffic will be:

$$L = N_u i \cdot \&ui + N_{u2} \cdot s_{u2}. \qquad (5.3)$$

Here $N_u i$ and N_{u2} are the number of update messages generated by machines that respectively belong or don't belong to the host selection candidate set. Similarly, $s_u i$ and s_{u2} are the service times needed to process host update messages that respectively belong or don't belong to the host selection candidate set. These quantities are given by:

$$\begin{aligned} N_{ul} &= N_g \cdot f_{it} \\ sui &= < + {}^{\pounds L \wedge \pounds} > \\ N_{u2} &= (N - N_g) \cdot f_u, \\ s_{u2} &= s_u^o + c_2. \end{aligned} \qquad (5\text{-}4)$$

Here $s°_u$ is the "basic" service time needed by the server to process an update message, Ci is the time required to traverse a single link in the ordered list of host candidates, and c_2 is the time needed to check for and discard an update that does not fall within the candidate set. These constants should actually be taken relative to a single constant that represents processor speed; we will use $s°_u$ as the reference constant instead. Measurements taken from the implementation described in Chapter 7 indicate that reasonable values to assume for a 1 Mips processor are 10 milliseconds for $s°_u$, 0.005 • s£ for ci, and 0 for c_2 (in comparison to s£). Thus, Equations 5.5 turn into:

$$\begin{aligned} N_{ul} &= N_a\text{-}fi, \\ s_u i &= <\bullet (1 + 0.0025 \cdot N_g), \\ N_{u2} &= (N\text{-}N_a)\text{-}f_u, \\ s_{u2} &= s_u^o. \end{aligned} \qquad (5.5)$$

An important parameter to decide is the size of the host candidate set, N_g. We want a size that will be just large enough to satisfy most bursts of host selection requests. For

the purposes of our estimate, we shall take N_g to be equal to N_r. This is sufficiently large for any reasonable-sized system.

Given server loads for both selection requests and update messages, we can calculate the aggregate average server CPU load as their sum:

$$l = l_r + l_u. \tag{5.6}$$

We estimate the queueing delay at the server by using the waiting time bound for a $M/G/1$ queueing model given by the Pollaczek-Khinchin mean-value formula[7]:

$$D_q = \frac{l \cdot (1 + C_u^2)}{2 \cdot (1 - l)}, \tag{5.7}$$

where C_u^2 is the coefficient of variation of the service time distribution. We will treat s_{u1} as being bounded by a constant, $s_u^o + c$, and approximate C_u^2 by 0. Thus:

$$D_q = \frac{l}{2 \cdot (1 - l)}. \tag{5.8}$$

From this we estimate the average response time to a host selection request with:

$$t_{req} = t_{msg} + D_q \cdot s + s_r, \tag{5.9}$$

where s be the average service time for an arbitrary message:

$$s = \frac{l}{N_r + N_{u1} + N_{u2}}. \tag{5.10}$$

Figure 5.1 shows how a central scheduler can be expected to perform, to first order, under various perturbations of the quantities just discussed. Interesting things to note are:

- The server is quite capable of handling systems of 100 to 200 machines in all cases, even when the server is running on a 0.5 Mips machine (corresponding to a base service time of 20 milliseconds for both s_r and s_u^o).
- If the server is run on a "standard" 1 Mips processor (corresponding to service times of 10 milliseconds), then between 700 and 800 machines can be handled easily, each of which is generating one remotely executing *non-distributed* applications per minute. If we are willing to run on the server on a 2 Mips machine (which corresponds roughly to a SUN-3 workstation), then these numbers double to 1400 to 1600 machines. Thus, investment in a single 2 Mips machine would allow a system of 1500 1 Mips workstations to live comfortably with a single central scheduling server. On the opposite end, even if the system consists of 0.5 Mips machines, the server should be able to service a system containing between 300 and 400 machines.
- If we make the average "degree of distribution" of applications equal to $d = 5$, then the server running on a 1 Mips processor should still be able to accommodate systems containing up to about 500 machines. Doubling or halving the processor speed of the server machine yields the corresponding changes in system size of 1000 and 250 machines. Note that this assumes that the *average* remotely executing application contains 5 programs as part of it. Experience with the system described in Chapter 7 indicates that, so far, this is a very optimistic assessment of how prevalent distributed applications will become.

[7]See the appendix of [Kle75] and Chapters 1 and 2 of [Kle76] for a discussion of the queueing theory we make use of here.

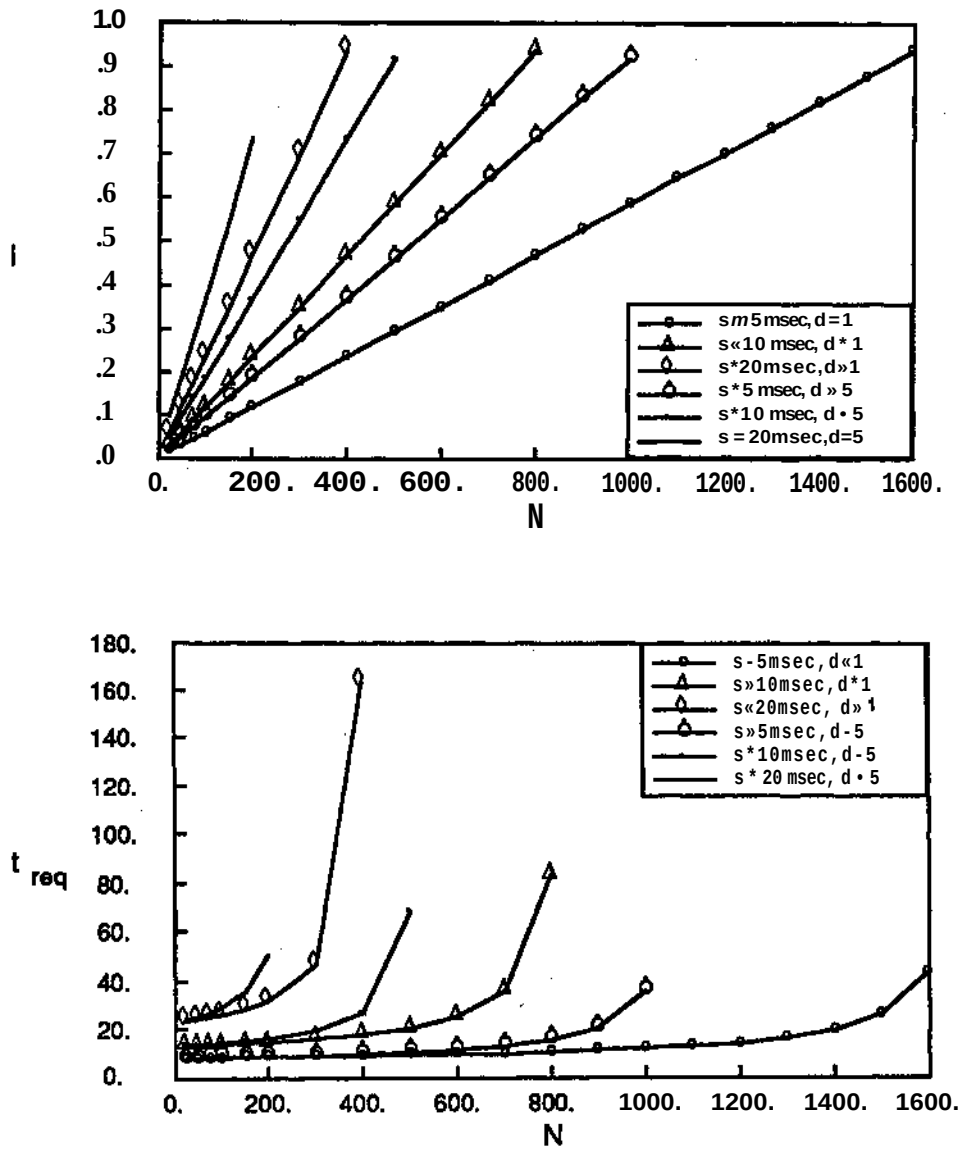

Figure 5.1: Central server performance with all hosts updating it. In all cases: k = s_r = <, fi = fu = 6/min, and t_{msg} = 3 msec.

- Not surprisingly, if the host candidate set maintained by the server is relatively small, then the update activity of idle machines becomes relatively small and unimportant (i.e., the results are insensitive to f_i). The update load is determined by the time to discard updates from busy machines rather than the time required to deal with idle machines' update messages. This observation will be used next to redesign the server to remove this part of its load.

5.3.2 Restricting the Number of Updaters

We can substantially lower the message traffic to the server by realizing that we are only interested in update information from those hosts that are likely candidates for remote execution, namely, from the lightly loaded hosts that comprise the host candidate set described previously. An update host group can be created whose members send updates to the server, and all other hosts in the system don't.[8] Membership in this group is determined in a decentralized manner by using multicast to send out a load cutoff value. Hosts with a load below this value join the update group if they are not already a member of it. Hosts with a load above this value leave the update group if they already a member of it. The server adjusts the size of the group by changing the cutoff point, choosing a group size, N_g, that is suitably sized to handle reasonable variations in the host selection request frequency, as before.

In order to avoid bothering busy hosts with cutoff value messages, we can create a multicast group containing those hosts that have some idle resources. In order to determine the appropriate cutoff change increment, the server can have all hosts in the "idle resources" group periodically update the server at low frequency in order to provide it with an estimate of the system's overall load distribution.

To obtain estimates of how this new design affects the size of system we can handle, we can use equations 5.1 through 5.10 with N_{u2}, which is the number of update messages generated by machines that do not belong to the update group, set to 0. Figure 5.2 shows the equivalent results of Figure 5.1 with these changes taken into account. The most important thing to note is that the server is now able to handle in excess of 1000 machines in almost all cases. The only case that can't be handled is that of running on a 0.5 Mips processor and having to deal with distributed applications of degree 5. For our "canonical" case of running the server on a 1 Mips machine and dealing with non-distributed applications, the *server* can deal with systems up to about 5000 machines in size. If we invest in a 2 Mips server machine, then systems of up to 10,000 machines can be handled. At this level of size, the server will almost certainly no longer be the bottleneck. For example, it is doubtful whether any current local area network could sustain the traffic generated by several thousand machines that are all attempting to remotely execute programs once a minute or so. Thus, it seems reasonable to claim that the centralized scheduling design we have devised should never present a performance bottleneck for (simple) host selection to find least loaded hosts.

If multicast is not available, then we can still employ a variation of the same approach, except that larger group sizes must be employed. Instead of multicasting cutoff values, machines would periodically have to "check in" with the server to determine the current cutoff values.

5.3.3 Distribution of Remote Execution Requests

While it is reasonable to assume a Poisson distribution for requests from different applications and updates caused by the actions of different applications, one might worry about how distributed applications will affect the distribution of requests. As discussed in

[8]Members of the group would be the program managers of each host.

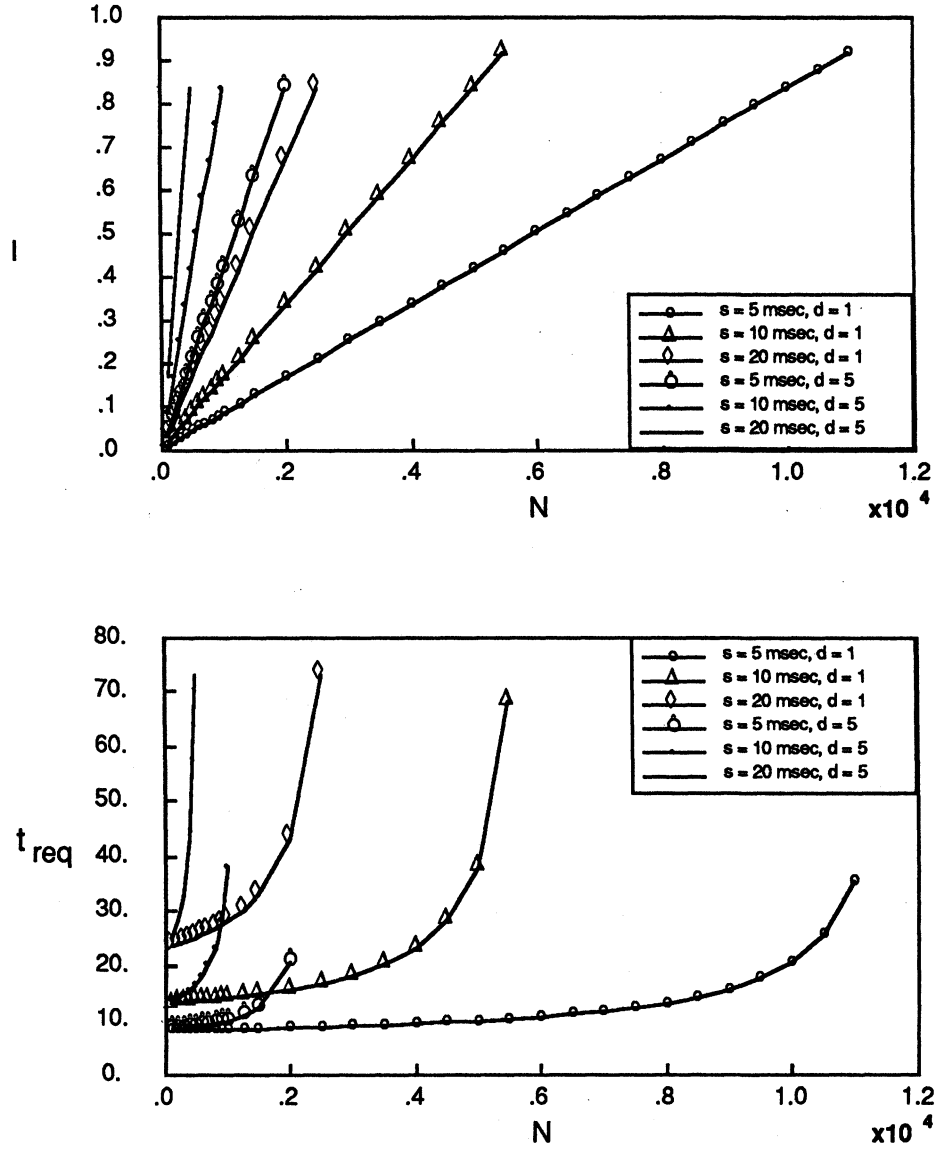

Figure 5.2: Central server performance with a restricted updater group. In all case
1/min, $s_r = s_u^o$, and $f_i = f_u = 6$/min, and $t_{msg} = 3$ msec.

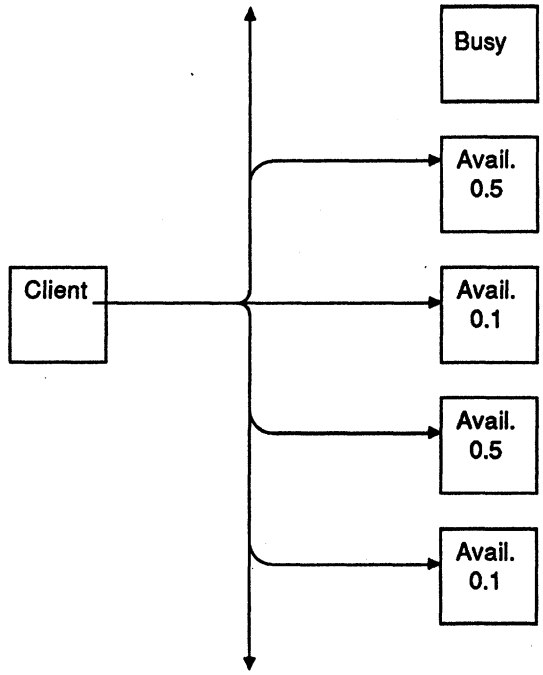

Figure 5.3: Querying the system for status information.

5.4.1 The Basic Model

A fully decentralized design requires that every machine perform its own host selection actions. This can be done either by having every machine keep track of the global system state continually or having it query the system for state information as needed. The former approach is essentially an *advertising* approach in which machines generate update messages, as in a central design, and broadcast these to all machines. The problem with such an approach is that every machine, including busy ones, must keep track of all updates. If a machine crashes, then it must also somehow reconstruct the state information it has lost.

In contrast, in a *query*-based approach only those machines interested in host selection must worry about state information. If queries are multicast to a group containing only machines with idle resources, as shown in Figure 5.3, then busy machines never participate in the process at all. Thus, the overhead of decentralized scheduling is incurred by lightly loaded machines and machines interested in remote execution or migration. In particular, the overhead of generating a status reply message to *every* host selection request is incurred only by lightly loaded machines that can absorb the overhead into their idle time. Additionally, there is no failure recovery procedure since state information is obtained only at the time it is needed.

The basic model we shall use, then, is one in which a client interested in obtaining a host selection sends a multicast query requesting current state information. It receives replies back from all willing candidate machines. The request may contain minimum resource requirement specifications to exclude some replies from being generated in the first place. The replies are then used by the client to make a host selection.

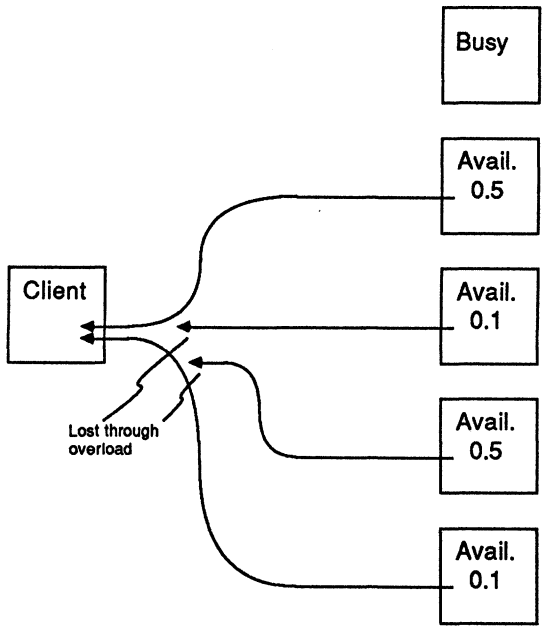

Figure 5.4: Reply set overload.

5.4.2 The Overhead of Multicast

An important question for our designs is how much overhead the use of multicast actually imposes at machines that are not members of the intended receiver group. Hardware support in the network interface of a machine can make the overhead close to zero.[10] However, many network interfaces only support broadcast, requiring that multicast group membership be determined in software.

The implementation described in Chapter 7 indicates that a 1 Mips machine can reasonably be expected to need at most 0.4 milliseconds to discard a multicast packet not destined for it in software. Under our standard assumption of standard one selection request per machine per minute, a system of 1000 machines would incur an overhead of about 0.7% on each machine for discarding multicast packets. In contrast, the overhead of providing round-robin scheduling on a local machine for the implementation mentioned is about 1.5% of the CPU. Given the increasing prevalence of hardware support for multicast, we will henceforth ignore the question of inherent multicast overhead.

5.4.3 Reply Sets

The principal problem with a query-based design is that the enquirer may receive a very large number of replies. This can overload the hardware and local operating system, as described earlier and depicted in Figure 5.4. It also prolongs the time required to perform a host selection since the selector must process at least some of the reply messages before being able to obtain a selection. In contrast, a central server can process and order update

[10] An example of a current network interface that provides hardware multicast filtering is the one in the SUN-3/75 that is based on the Intel 85286 chip.

messages more-or-less separately from selection requests.

If we assume that network interfaces have difficulty handling back-to-back messages on a network, then we must resort to artificially imposing random delays to reply messages in order to spread messages over a longer time interval. The choice of average delay value that each machine should use will depend on the number of replies expected *and on the reliability with which replies must be received.* If we must reliably receive all replies, then they must be spread over a considerably longer time interval than if a few may be lost. Both in order to minimize delays and to have a predictable average random delay interval to use, we must control the size of the *reply set* to be some small, relatively fixed number. A small reply set size is also necessary to avoid locking up a significant fraction of a machine's packet buffering capacity, which could lock out other activities from the network and cause timeouts of unrelated network communication transactions.

Measurements described in Chapter 7 indicate that a 1 Mips machine can process multicast query replies in about 0.5 milliseconds and that remote machines can process and respond to a status query message in about 8 milliseconds. If we assumed that packets arrived back-to-back with no intervening delays then a host selection procedure could process about 84 replies in 50 milliseconds, which corresponds to the response times we would expect to obtain from larger systems using a central scheduling server. Depending on the type of network interface available and the level of reliability desired we must stretch the time needed to receive (and process) query reply packets by a factor that might range anywhere from a value close to 1 to values greater than 10. Even a factor of 2 to 4 would increase host selection response times to between 0.1 and 0.2 seconds (which are still quite acceptable), or alternatively, decrease the number of replies that can be processed to between 20 and 40.[11] Experience with the implementation described in Chapter 7 leads us to argue that reply sets of more than about 40 are undesirable both from a processing time and a buffering point-of-view. However, these values are hardware dependent in the sense that more memory and higher network bandwidth will allow larger reply numbers.

The problem we face with controlling the size of query reply sets is essentially one of gathering the necessary global state information to determine which machines should be in the reply set and then propagating that information to those machines. This must be done on a continuous basis since the reply set continually changes. Unfortunately, in a decentralized design there is no focal point at which such information can be gathered and disseminated. The only alternatives are to multicast the necessary state information to all possible reply set candidates or to have idle machines periodically perform the functions of a central server. In the former case, the candidate machines are implementing the equivalent of a replicated central server. In the latter case, idle machines must periodically serve as a "focal point" for gathering and disseminating information about reply set membership to the set of all possible candidate machines, namely, the set of all machines willing to accept remote execution and migration requests. Use of a random delay interval by each idle machine to decide when it should next perform "focal point duties" is one means by which allocation of these duties can be performed in a decentralized manner.

How often idle machines "pop up" to recalculate reply set membership also affects how large a reply set size must be used. If some idle machine is always controlling the reply set, then we have the equivalent of a (migrating) central server. If reply set membership is controlled only periodically, then a sufficiently large reply set must be used to allow responses to all anticipated queries that will occur before the next recalculation. Unfortunately, the reply set size cannot be allowed to increase beyond our designated value of about 40 irrespective of how large the size of the system is.

There are several different approaches to controlling reply set membership:

[11]Note that even if the network interfaces used are able to handle arbitrary numbers of back-to-back packets, we would still experience some level of "dispersion" due to contention for the network by all the repliers.

- A load cutoff value could be calculated and sent to the "idle resources" multicast group. The problem with this approach is that it does not scale well with system size, as mentioned.
- A larger "reply set" could be tolerated if only some subset of its members reply to query messages. This approach would require that reply set members communicate with each other in order to determine who should reply to query messages. More specifically, reply set members would have to determine an ordering on themselves so that only the least loaded members reply. The level of communication among members would be determined by the rates at which machines enter and leave the set and the rate at which members' loads change with respect to each other. Since all communication is between machines that have (by definition) idle resources, the overhead of performing this ordering is not significant, except in determining an upper limit on group size at which the idle resources are exhausted by the communications.
- Let the "pop-up" server query a larger group and then multicast a membership message to the group indicating which machines should join the smaller reply set. In order for this approach to work, the pop-up server must run frequently enough to refill the reply set before a burst of host selection requests can empty it.

Since the principal goal we are after is simplicity of design (given the viability of centralized designs), we do not really consider the last two approaches to be reasonable alternatives. In order to achieve a *simple* load cutoff approach, we need larger reply sets that allow simple and infrequent recalculations of an appropriate load cutoff value.

5.4.4 Random Probe Sets

To achieve our goal of simplicity, we must find some additional means by which to expand the reply set size. If we are willing to accept host selections that are not "perfect" in the sense of being the "absolutely" least loaded host in the system, we can have an enquirer only examine the first n replies of a reply set of size m, discarding the rest. Response times and buffering requirements would thus be determined by the value of n, whereas the reply set size would be a much larger value m.

The error introduced by this approach can be reduced by having reply set members weight their random delays by a factor that reflects how good a candidate they think they are. For example, the delay might be constructed by taking a a uniformly distributed random variable and multiplying it by a standard delay interval and the machine load, normalized to a value between 0 and 1. The rest of this section will be devoted to examining how well this scheme can be expected to perform and how it relates to another scheme that was described in Chapter 2, namely the Random Probe Set scheme.

As discussed in Section 2.4.2, the Random Probe Set scheme relies on homogeneity assumptions to obtain a representative load sample from the system using only a small sampling size. The approach outlined above can be viewed as a means of achieving the same effect in a less homogeneous environment. To ensure that probes obtain a representative sample of loads of lightly loaded hosts in the system we exclude all hosts that *aren't* lightly loaded from the set of hosts that are probed. By ordering the reply set and excluding uninteresting candidate machines from the probe space, we actually improve on the Random Probe Set approach by "loading" the probe set to contain only good choices.

Since replies are approximately ordered by quality and only a few replies need be looked at to obtain the desired sampling accuracy, we can use a fairly crude load cutoff scheme to limit the number of replies generated.[12] One very simple cutoff approach we can use if the system's aggregate load does not vary rapidly is a scheme whereby all machines with a load less than p respond to queries. The enquirer then optionally sends an "adjustment"

[12]Note that for systems with only 100 or so machines, we can completely ignore the load cutoff issue.

message out if too many or too few replies were received. If there are not many idle machines, then the rate at which machines cross the cutoff load level will be determined by the starts and terminations of remotely executing programs and the average number becoming idle per unit time will be equal to the average number becoming busy. If there are many idle machines, then the size of the reply set will not be significantly affected by the rate at which programs are remotely executed since there will always be many idle machines available. In this case, an additional mechanism must be employed to reduce the reply set size. Since load is no longer a significant distinguishing factor, we can employ random criteria to delineate machines. For example, if the reply set is j times larger than desired, then idle machines can be instructed to pick a random integer between 1 and j and only reply to query messages if the random integer obtained is 1. New "load cutoff" messages would thus be sent out whenever the total number of idle machines in the system changed significantly.

Ignoring contention issues for the moment, the size of system that can be handled by this approach will be determined by maximum number of replies, m, we are willing to receive or discard and by the relationship between m and the system size N. If we assume that the time needed to discard a query reply unexamined (but still received in the machine) is similar to the time needed to filter multicast messages in software, then a 1 Mips processor can discard about 125 replies during the 50 milliseconds we previously suggested as a target response time. Taking into account the 8 milliseconds needed to send a query message and have it processed by the machines in the reply set reduces this number to about 100.

If we wish to maintain a relatively stable cutoff value p in the face of limited idle machines, then we must make the reply set size large enough to handle reasonable variations in host selection request frequency. From previous arguments, this implies that we must make m equal to roughly the value of N_r. For non-distributed applications (d = 1), this implies a system size of about 600. Note that since we are willing to occasionally incur a small extra delay, then we can handle unusual bursts of host selection requests by having clients send a second query message out if their first one does not return a reply within a short period of time, say, 20 milliseconds. This second query can specify a temporary cutoff value p' that will result in additional replies.

If there are many idle machines available, then m must merely be a large enough fraction of N to allow a non-zero number of replies to be generated by the random integer approach described almost all of the time. For example, if m is made to be at least 10% of N, then N can range up to 1000 while still maintaining virtual certainty that one or more replies will be returned to every host selection request. Since the main premise for providing remote execution facilities is that there are usually many idle resources available, this latter estimate for appropriate reply set size to use should be the more appropriate one.

An important capability that must be provided for this approach to work without undue interference being caused to the operating system is a means by which the operating system (and kernel) can be told when to discard queued multicast reply packets. Otherwise, a machine may be clogged with unneeded multicast packets long after the scheduling client has examined the first n packets to a status query, made a scheduling decision, and gone on to other tasks.

5.4.5 Contention

A problem faced by designs with decentralized control is that of contention. Since host selections made are not evident to everyone until the time at which a remote execution request is actually sent to some host and accepted there, there is a time interval during which the global state information available to other host selection clients will be out-of-date. Any new host selections initiated during this time interval will see a global state that

does not reflect the pending host selection of the first one and hence, might pick the same host as their remote execution candidate. In the case of many idle machines, the random nature of the reply set membership should usually avoid the problem.

Poor scheduling decisions can be detected by having a program load request include the load it is expecting to see on the machine the request is being sent to. If the actual load is significantly worse, then rescheduling can be performed *before* the cost of loading has been incurred. If the load request also includes an alternative selection candidate and its expected load, then rescheduling can consist of simply forwarding the load request.

One must now worry about cascading effects where rescheduled load requests conflict with the second selection made after the one being rescheduled. The forwarding trick only helps if the likelihood of contention is small enough that the likelihood of two selections occurring between a host selection and its associated remote execution or migration request is fairly small. If this is a problem, we can have alternatives be randomly selected from an appropriately large set of the best candidate hosts. This will effectively eliminate contention. The same arguments that make the Random Probe Set model work in the first place will also ensure that this additional "shuffle" of selections should not significantly affect the goodness of the scheduling choices made.

To obtain a crude estimate of the probability of contention if there is a significant load spread (implying that closely spaced queries will likely result in identical host selections), we will assume a random arrival rate for query messages and introduce a new parameter, the length of time between a client's query message for state information and the subsequent remote execution or migration request message to the host it has selected, t_r. Given our assumption of a random arrival rate for query messages, the probability of at most x query messages arriving within one second after the first one is given by the cumulative Poisson distribution,

$$F(x, \lambda) = \sum_{k=0}^{x} e^{-\lambda} \frac{\lambda^k}{k!}, \tag{5.11}$$

where λ is the average number of arrivals per second. In our case, we are interested in the probability of messages arriving during the time interval t_r; hence, we must replace λ with $N_r \cdot t_r$. Furthermore, we want to know the probability of *at least* one message occurring during that time. Hence,

$$\begin{aligned} p &= 1 - F(0, N_r \cdot t_r) \\ &= 1 - e^{-N_r \cdot t_r}. \end{aligned} \tag{5.12}$$

Figure 5.5 shows how the probability varies for various values of t_r and various system configurations. Since rescheduling is a fairly cheap operation to perform and we assume that the common case is for a system to have fairly many idle machines and we are interested in keeping our design as simple as possible, we argue that one should simply ignore the problem of contention for most systems.

An important point to note however, is that host selection clients should be careful of how much time they spend between making a host selection and subsequently sending a remote execution or migration request to the selected machine. For example, consider what might happen if remote execution is structured so that clients first query the system for available machines and then search for the correct program file to load. If the latter operation involves traversing a search list of possible locations within a (remote) file system, then the time required to find the correct program file may be considerably longer than the tens of milliseconds assumed for t_r above. This problem can be exacerbated in heterogeneous systems in which the search list may imply looking at multiple locations for each list entry—one for each machine type in the system.

An issue closely related to contention is that of caching global state information. If we cache state information for periods much longer than the average time interval between

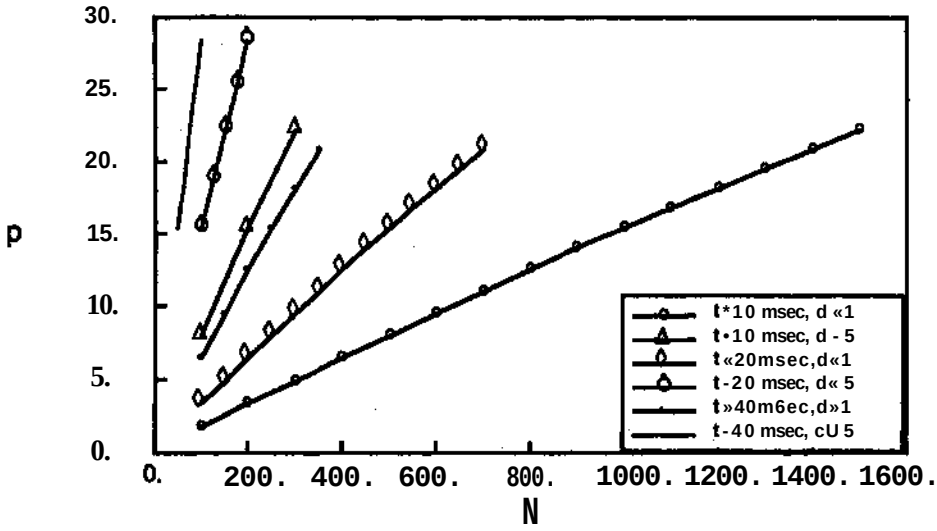

Figure 5.5: Probability of contention between host selection actions. In all cases: $k = 1$.

host selection requests generated in the system, we will run into the same contention problems described before. As described earlier, this should not be a problem if there are many idle hosts in the system. However, if the state of the system does not provide host selection with several essentially equivalent host selection candidates, then caching between applications on a single host may actually be counterproductive in larger systems (where the time between requests will be smaller). Distributed applications, on the other hand, may generate a batch of requests in short succession and for these caching would be of greater interest.

5*5 Summary and Conclusions

In this chapter, we have described two scheduling designs that perform well, are highly available, and scale to large system sizes. We have shown that if efficient multicast communication facilities are available, centralized and decentralized scheduling designs can be made comparable in availability. Arguments presented in this chapter and quantitative assessment of the implementation described in Chapter 7 indicate that both centralized and decentralized scheduling designs should be able to achieve comparable performance levels. That is, both design approaches should provide host selection candidates of comparable quality and host selection response times of comparable length. We have also shown that although centralized designs can be constructed that don't rely on broadcast communication, they can be considerably improved when such facilities are used. In contrast, multicast, or at least broadcast communication, is a prerequisite for a decentralized design in our environment.

The centralized design we have presented is extremely scalable, being able to handle a system of thousands of machines, at least as far as the scheduling server is concerned. This is achieved by limiting status update message traffic to only those machines that are likely host selection candidates by creating an update group whose membership is small and limited by broadcasting a membership load cutoff value.

Our decentralized design is based on the same idea of sampling a small number of hosts in the system to obtain an estimate of the system's load distribution as is used in the Random Probe Set model proposed by Eager et al[ELZ84]. The Random Probe

Set model relies on a homogeneous load and selection request distribution to ensure that small sample sizes will provide sufficient information about lightly loaded hosts. The workstation environment we must deal with may not satisfy these assumptions about load and selection request distribution. To achieve the same effect, we use multicast to create a potential "reply set" containing only good host selection candidates. Host selection is performed by multicasting a query message to this reply set and only processing the first *n* replies sent by the reply set's members. The rest of the replies are either discarded or lost in the "flood" of replies that each query message generates. By relying on a statistical algorithm, we are able to avoid the problems of reliably receiving a large number of reply messages within a short period of time. This approach also limits the amount of time required to perform a host selection by limiting the number of replies that must actually be processed.

The size of the reply set, m, is determined by the conflicting desires to keep the reply traffic load low—every reply causes some overhead on the network and discarded replies cause overhead at their receiver—and the need to be able to control reply set membership by some simple means such as a periodically calculated load cutoff value. These considerations, together with limits imposed by contention between distributed host selections under some circumstances, limit our decentralized design to handling systems of at most a thousand machines or so. In exchange for this limit on scalability, which we argue will not in fact be the real limit on system size, we obtain a simpler design. There are no failure recovery procedures, no failure detection requirements, and no need to migrate a scheduling server between idle machines if a dedicated server machine isn't available. The last point becomes especially important if the system does not support migration, in which case the availability of a central server becomes more difficult to maintain unless several dedicated server machines are available.

An important point to note in anticipation of special situations, or extension to an internet environment as discussed in Chapter 8, is that nothing prevents the creation of combined approaches in which a central server is invoked, possibly temporarily, as an optional or added service. As long as the central server is constructed to realize that it is but one of many scheduling entities, it will be able to transparently fit into a decentralized design.

Chapter 6

Additional Scheduling Issues

6.1 Introduction

The previous chapter dealt with the basic problem of finding lightly loaded machines. In this chapter, we will examine several global scheduling issues that are not essential to the support of preemptable remote execution, but that can affect the manner in which the system treats users and their programs. Specifically, we will address issues of fairness and load balancing.

Fairness is a term that can have many meanings. We shall define it to mean that the share of the system's resources received by each "entity" is the same. This still allows for a variety of different kinds of fairness. For example, there can be fairness among users, meaning that each user in the system receives an equal share of the system's resources. There can also be fairness among programs, meaning that each program in the system receives an equal share of the system's resources.

Another criterion to consider is the time interval over which fairness is enforced. Equal resource shares can be provided over a period of time ranging from milliseconds to days. If fairness among users is the principal goal, then it may be quite reasonable to ensure fairness only to the level of equal access averaged over a period of hours and the execution of several programs. On the other hand, if every (remotely executing) program in the system is to receive an equal share of the system's resources, then fairness must be enforced over time intervals measured in seconds or less.

Yet a third consideration is that some applications must have a guaranteed number of resources to function correctly, such as benchmark programs. Distributed applications that perform internal scheduling decisions based on load assumptions such as the absence of competing load sources fall into a similar category. While such applications will still function correctly if their load assumptions are incorrect, their overall runtimes may be significantly increased due to inappropriate scheduling decisions made for individual subprograms. Similarly, some types of distributed applications run at the speed of their slowest subprogram, implying that their subprograms should all receive equal resource shares in order to minimize the application's turnaround time and minimize the amount of time that subprograms are idly taking up system resources while waiting for slower instances to complete.

With these considerations in mind, one must decide what kind of fairness to provide and what level of fairness to enforce. Depending on what goals are desired and what kinds of applications the system is expected to run, different, possibly conflicting policies will be appropriate. For example, consider a situation in which user A is running a distributed application consisting of 5 subprograms, each running on a different machine. User B now wishes to remotely execute a single program that ends up being scheduled on one of the machines on which user A is running a program. If we wish to provide fairness among

users, then A's subprogram should receive a much smaller share of the machine's resources than B's program or perhaps should even be migrated to another machine. But if A's application runs at the speed of its slowest subprogram, then this strategy would actually penalize A far more than intended, aside from making poor use of the system's resources. Additionally, one might ask whether B really cares that A is using resources on 5 machines and wouldn't be satisfied with receiving equal shares with A's subprogram on the one machine they are both using.

What kind and level of fairness to provide is a *policy* issue that we will not address in this thesis. Instead, we argue that fairness issues can be ignored if we only seek "statistical" fairness between users. "Statistical" fairness is defined here to mean that users obtain equal shares of the system's resources, averaged over the long run of many programs executed, assuming that all users generate the same load, averaged over the long run.

We will also describe a particular global scheduling policy and load balancing scheme that can be used to bound the degradation in runtime a distributed application can experience due to unequal loads on various machines. Furthermore, for applications whose subprograms are independent of each other and, hence, don't communicate with each other (implying that they don't have to be run concurrently), we will claim that one cannot do much better than our scheme without a priori information about the programs to be run, such as their expected completion times. The scheme described will deal *only* with CPU resources, ignoring the question of fair access to other resources such as memory, secondary storage, and special devices such as hardware floating point support facilities.

6.2 Fairness Among Users

If our goal is to provide users equal access to the system's resources over the long run—meaning multiple program executions and equal long term user workloads—then the issues of fairness and load balancing need not be explicitly considered—the system will provide them automatically. We claim that this approach is sufficient for most purposes, that more stringent forms of fairness and attempts at load balancing are useful mainly for exceptional cases such as distributed applications whose overall performance depends critically on the provision of (roughly) equal resource shares to each subprogram. An example of a system that has successfully taken this approach in a time-sharing environment is UNIX, which makes no attempt to regulate the resource usage of (equal priority) users except to provide a round-robin scheduling discipline for access to the CPU. Note also that users always have access to a minimum share of the system's resource, namely, to those on their own machines. Thus a long term fairness policy will never completely starve a user even during short term fluctuations in the system's load distribution.

In most cases, the issues of fairness and load balancing never come into play at all. For example, as long as the system is lightly loaded, so that remotely executing programs never have to compete for the same resources against each other, complete fairness and an "optimal" load distribution are obtained automatically. Since we assume the existence of plentiful idle resources as the normal and predominant case, users should only rarely experience any other situation than this. Conversely, if the system if fully loaded, then fairness among users and load balancing are also irrelevant in that there is no reason to remotely execute or migrate programs in the first place. The optimal scheduling policy is to execute every user's workload on his own machine and to do no global scheduling at all.

If users consistently execute different numbers of programs, then fairness among users will not be obtained. This may not be a problem in the sense that the system's user community may have some externally defined hierarchy of privilege that regulates access to resources. It may also be acceptable if differences in submitted workload are not permanent—for example, if users occasionally generate a greater workload for a few days at a time.

Naturally, fair access to any given machine's CPU resources can be provided by round-robin scheduling each machine among its *users* rather than among its programs. This would place a cap on how much of the system's CPU resources any given user could acquire by submitting additional programs for execution.

Equalizing resource shares among users any more than this would require either migrating programs around or altering the resource snares given each user on individual machines. In either case, we face the task of gathering the necessary state information about what is running where, deciding whether or not to manipulate any user's programs, deciding which programs specifically to put in a lower priority class or to migrate elsewhere, and then actually taking the chosen action. Furthermore, we must worry about the stability of any preemption policy used. If programs are preempted from a machine to free its resources and are allowed to migrate back to it later on, then a form of "thrashing" may occur in which many cycles are wasted continually preempting programs. Conversely, if machines are kept idle in anticipation of servicing requests from "preferred" users, then we end up idling resources needlessly.

An alternative approach is to provide an *explicit* "background" priority class that users can place additional workload—such as large distributed applications—into. This priority class would receive resources only when no programs of higher priority are runnable, implying that it would not interfere with the execution of users "normal" workloads. ISince inclusion in this priority class would be explicit, using this approach would not require the system to gather and maintain any additional global state information or to perform any additional scheduling decisions. However, this approach could only be used for applications that are relatively insensitive to variations in the resource shares they receive, since a background priority class would be subject to sudden, major changes in the number of resources available to the programs running in it.

All things considered, we conclude that extension of global scheduling facilities to explicitly account for fairness considerations is uncalled for. However, this conclusion only holds true as long as there are no hard resource limits to contend with, such as memory constraints. If such limits exist, then we must control *acquisition* of resources since migration can't create new resources and destruction is the only other form of preemption from such resources. One could imagine various schemes for allocating non-sharable resources, such as market-based schemes, etc. However, such considerations are beyond the scope of this thesis and may not even be an issue in most systems. If demand-paged virtual memory is available, then the "standard" resources of processor and memory available on most machines have no hard limits on sharing.

As a final point, we will note that implementation of any algorithms that require information from every host in the system will almost certainly have to rely on centralized servers to gather this information since decentralized designs must somehow limit the number of responses they receive from their queries of the system's state. Thus, one of the situations where a centralized scheduling design may be more appropriate than a decentralized one is when specialized resource limits and a sufficiently high system load force a non-trivial fairness policy to be implemented.[1]

6,3 Equalizing CPU Shares

A class of applications for which "statistical" fairness among users is not sufficient is that of distributed applications whose overall runtimes are sensitive to the resource shares each of their subprograms receive. An important subclass of this is applications whose subprograms are independent of each other and, hence, don't have to communicate. Frequently, these applications run at the speed of their slowest subprogram, often consisting of a series of one or more parallel computation steps followed by synchronization of the results. This

[1] A discussion of specialized resource limits can be found in Craft [Cra83].

class includes many traditional artificial intelligence and database applications, in which multiple search spaces or data can be processed in parallel. Other examples of applications include dynamic programming, zero finding, and matrix multiplication.[2]

In this and following sections, we will describe a global scheduling policy and load balancing scheme whose goal is to minimize the effect of slowing down one or more subprograms on the entire application's runtime by ensuring that each subprogram receives at least a share of CPU cycles that approaches the average share received by all remotely executing programs in the system. Our global scheduling policy is based on the same "least loaded host" selection algorithms described in Chapter 5, with the restriction that the selection load metric be the number of guest programs on each machine. Furthermore, preemptive migration of guest programs from any machine that initiates significant local—and, hence, higher priority—activity is *required.* Significant local activity is defined to mean whatever level of resources we are willing to forego a guarantee on for guest programs. For example, if preemption is to occur whenever local activity exceeds 10% of the CPU, then we can only assure guest programs of access to 90% of any given machine's CPU cycles, ignoring the cost of preemptive migration for the moment. We also assume that each machine employs a round-robin scheduling discipline for regulating guest programs' access to the CPU, so that guest programs on a machine all receive equal shares of the CPU even in the short run.

Performing host selection based on program count (the number of programs running on a machine) results in most machines having close to the same number of programs on them. If, on average, there are N remotely executing programs running in a system with M "available" machines (i.e., no significant locally initiated activity), then each available machine will have close to $\hat{n} = \lfloor N/M \rfloor$ guest programs on it. Since the scheduling algorithm always places new programs on machines whose number of guest programs is low, close means that many machines will have $\hat{n}$ or $\hat{n}+1$ programs on them. Machines will have fewer than $\hat{n}$ programs on them whenever a program terminates and hasn't been replaced by a new one yet. Some machines will have $\hat{n}+1$ programs running because $\hat{n} \cdot M$ is less than N. However, machines can only have $\hat{n}+2$ programs running on them if a temporary fluctuation in load creates an additional M programs beyond the average load of N, since no machine with load $\hat{n}+1$ will receive a new program until all other machines also have a load of at least $\hat{n}+1$. That is, as long as the system is within M programs of the average load N, we can guarantee each subprogram a fraction $1/(\hat{n}+1)$ of the non-preemptable CPU cycles on its machine. Note that the average CPU share provided by the system to guest programs will usually be some fraction between $1/(\hat{n}+1)$ and $1/\hat{n}$.

The size of $\hat{n}$ will determine how likely the occurrence of a "significant" load fluctuation is. In general, the load must increase (temporarily) by a factor $1/\hat{n}$ over its average value in order to increase the load of any machine to $\hat{n}+2$. A more interesting measure of load variation is the effect of temporarily *doubling* the system's guest program load. This measure is independent of the size of $\hat{n}$. The disparity between machine loads caused by any increase in program load *less* than double will be bounded by 2 since no machine can receive more than twice the average number of of guest programs, $\hat{n}$, until all machines have.

6.4 Migrating Subprograms of Distributed Applications

To do better than is achievable just through appropriate host selections for remotely executing programs requires the use of migration. To see why this is the case, consider

[2]See, for example, Cheriton and Stumm's discussion of the problem of subprograms that must execute in "lock step" in [CS86].

the options available: We can either alter the CPU shares given each guest program on a machine or we can migrate guest programs among machines. Altering the CPU shares on a heavily loaded machine to favor one application's subprogram will penalize all other programs on the machine—even though these programs may in turn be subprograms of another application in the same predicament. Thus, we cannot accommodate more than one application by this means.

The key issue with migration is the determination of whether it will provide a net reduction in completion time for the subprogram being migrated. Note that our decision metric is *not* system throughput—which will be reduced if machines are fully loaded. Our goal is strictly to improve the overall runtime of the distributed application whose subprograms are being migrated. The justification, of course, is that the benefits of reducing the "waiting time" of a large distributed application's subprograms will outweigh the loss in CPU cycles incurred through migration.

Whether or not to migrate a program can be determined in a straightforward manner if we know the loads on both the source and destination machine, the number of cycles the program still needs to complete (its expected completion time under a given load), and the size of the program. Except for expected completion time, these quantities are directly available. Lower bounds estimates on completion time can be obtained by extrapolating the program's history into the future and assuming that it will run at least as long as it already has.

If we define the following quantities:

- the number of CPU cycles needed by the program for completion: c,
- the number of cycles per unit time available from a machine to a guest program if no other guest programs are competing against it: c_0,
- the fraction of c_0 available to our program on the source machine, where it is currently running: f_s,
- the fraction of c_0 available to our program on the destination machine, where it would be migrated to: f_d,
- the size of the program: s,

and assume that the cost of migration, in number of cycles needed on each machine, of a given program is:

$$c_m = a \cdot s + b,$$

where a and b are constants, then the program should be migrated if the time needed to complete the program on the source machine is greater than the time required to migrate the program plus the time required to complete the program on the destination machine:

$$T_s > T_m + T_d, \tag{6.1}$$

where

$$\begin{aligned} T_s &= \frac{c}{f_s \cdot c_0}, \\ T_m &= \frac{c_m}{c_0}, \\ T_d &= \frac{c}{f_d \cdot c_0}. \end{aligned}$$

We assume that migration runs at high priority, so that it receives all cycles on a machine while it occurs. Equation 6.1 can be rewritten as:

$$\frac{f_d \cdot f_s}{f_d - f_s} < \frac{c}{c_m},$$

which basically states that we need a sufficiently large ratio of program completion time to program migration time.

Chapter 7 presents measurements that indicate that the cost of migration is relatively small—on the order of a few seconds of CPU time on a SUN-2 workstation.[3] If we assume that the subprograms of distributed applications typically run for times of several tens of seconds or more (which they must in order to justify the initial cost of loading a new program), then migration will usually be justified and will be especially useful for dealing with the exceptional cases where subprograms end up on machines with more than $\hat{n}+1$ guest programs on them. Section 6.7 describes how a load balancer can be devised that continually monitors the subprograms of a distributed application and migrates those that reside on heavily loaded machines.

6.5 A Load Balancing Scheme

For distributed applications whose subprograms do not need to communicate with each other, a load balancing scheme can be devised that ensures even tighter bounds on CPU shares than the factor of 2 that can be obtained by the methods described thus far. Completely equalized CPU shares cannot be achieved because the cost of migration is nonzero and because a simplified migration algorithm must be used to decide when to migrate. The net result is that the turnaround time for a distributed application that is run with our load balancing scheme will be within 17%, plus the time to migrate each subprogram at most once, of that achievable if every subprogram of the application were to somehow receive exactly equal CPU shares. For the average case, when an application ends up on an equal number of lightly and heavily loaded machines, turnaround time will be within about 12% of the "equal shares" turnaround time.

6.5.1 A Simple Migration Scheme

There are two approaches one can take to equalizing CPU shares through migration: One can simply migrate subprograms from heavily loaded machines to lightly loaded machines as they become available or one can swap subprograms on heavily and lightly loaded machines with each other. Except for migrating off machines whose load is greater than $\hat{n}+1$, we cannot, in general, assume that lightly loaded machines will be available that aren't already running a subprogram of the distributed application in question. Thus, the former approach implies that we may have to wait for subprograms on lightly loaded machines to finish before other subprograms can be migrated onto them. The latter approach suffers from the problem that without a priori information we cannot estimate the runtimes of subprograms and, hence, cannot determine when the right time to swap subprograms among machines is. Although lower bounds on runtimes can be estimated by extrapolating subprograms' past history into the future, these are not sufficient for our needs, for we wish to swap subprograms only once. Without some estimate of an upper bound on runtime, we cannot determine how many times we would end up swapping subprograms and, hence, cannot estimate how much runtimes would be degraded just through the overhead of migration. The former approach is simpler to implement because it avoids the problem of guessing when to migrate.

We can calculate the difference in completion times between our simple migration scheme and a scheme that would somehow completely equalize CPU shares among all subprograms by considering an application whose subprograms run on $n-1$ heavily loaded machines and 1 lightly loaded machine. Actual applications correspond to several such "base" applications concatenated together so that the correct number of lightly loaded

[3] SUN workstation is a trademark of Sun Microsystems Inc.

and heavily loaded machines is obtained. Note that, on average, *n* should be 2; that is, we expect that, statistically, a distributed application should end up running on an equal number of machines with load *h* and load *h* + 1. (Remember that loads greater than *h* +1 should occur infrequently and should be dealt with by migration to a "normally" loaded machine in any case.)

Assuming that all subprograms have equal runtimes, the completion time for an application, if all its subprograms received equalized shares of CPU cycles must satisfy the relationship:

$$T_0^n + (n-1)\cdot \qquad = n \cdot t_{sp},$$

where t_{sp} is the completion time of a subprogram on a lightly loaded machine, A is the factor by which the more heavily loaded machines are slower, and *Tg* is the completion time for the application. That is, the work done on the lightly loaded machine and each of the *n* — 1 heavily loaded machines must equal the total work to be done, namely, the n subprograms of the application. Solving this equation for TJ^1 yields

$$T_0^n = \frac{n}{\Delta + n - 1} \cdot \Delta \cdot t_{sp}.$$

The completion time for the simple migration approach is

$$T_1^n = T_1^{n-1} + (t_{sp} - \frac{rpn-1}{\Delta}).$$

That is, the time to finish *n* subprograms by migrating *n* — 1 subprograms to the lightly loaded machine in sequence is just the time to finish n — 1 subprograms in the same manner plus the time required to finish what is still left to be done of the n-th subprogram. This can be rewritten as:

$$rp_1^n \quad \overset{\text{n-1}}{\underset{\text{Ar=0}}{\sum}} A_t \cdot t_{sp} \left(1 - \frac{1}{\Delta}\right)^k = \left[1 - \left(1 - \frac{1}{\Delta}\right)^n\right] \cdot \Delta \cdot t_{sp}.$$

From these quantities, we can compute the ratio of the difference in completion times to the equal shares completion time as

$$\delta = \frac{rp_n \quad rpn}{T_0^n} = \frac{1}{n} \cdot \left[\Delta - 1 - \left(1 - \frac{1}{\Delta}\right)^n (\Delta + n - 1)\right].$$

Plotting this difference for various values of n, and A values between 1 and 2, shows that it has a maximum of 17.2% at A = 2 and *n* = 4, as illustrated by Figure 6.1. For the average case of *n* = 2, when an application ends up on an equal number of lightly and heavily loaded machines, the maximum difference is only 12.5%.

6*5.2 Improvements

The load balancing scheme we have described could be improved by realizing that we can obtain an estimate for the runtimes of subprograms from the first subprogram that finishes

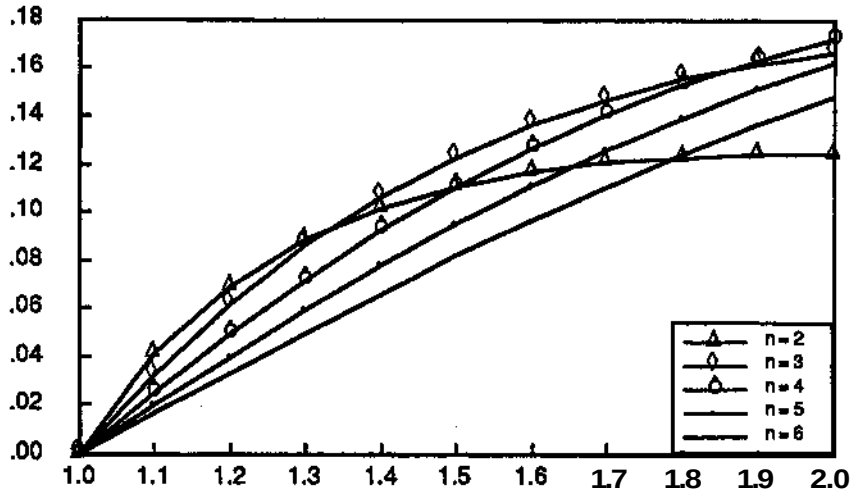

Figure 6.1: Ratio of difference in completion time to equal shares completion time as a function of the difference in machine loads, A, for various numbers of machines, rc.

and assuming that all subprograms have equal runtimes. We could use this information to attempt to migrate subprograms at the "optimal" time in order to implement the subprogram swapping scheme mentioned previously. While one can calculate whether or not the extra migrations will justify these swapping operations, it is unclear whether the extra improvement is worth the implementation effort. Note that this improvement comes into play only after the first subprogram of an application has finished. At this point in time, all subprograms on lightly loaded machines will be almost done and the improvement we obtain will depend on how many subprograms there are on heavily loaded machines *in excess* of the number that will shortly be transferred to lightly loaded machines anyway. If we assume that, on average, a distributed application will end up running on roughly equal numbers of lightly and heavily loaded machines, we will not obtain much benefit.

Another approach is to try to improve average case behavior by making assumptions about typical applications' subprogram runtimes so that a swapping migration scheme could be implemented immediately rather than after the first subprogram has finished. While this approach might well provide some improvements, it is again unclear whether it is worth implementing given the relatively good performance obtainable with the simple migration scheme—which is independent of application-specific assumptions about runtimes.

6.6 Comparison of Load Metrics

As mentioned in Section 5.2.2, processor utilization is a more accurate load metric than program count if we must take into account interactive programs, which have unpredictable CPU requirements and are many times left running in an idle state. If the interactive component of the system's workload can be separated from its compute-bound, or "batch" component, then program count provides as good a load metric for the batch component as does processor utilization (see below and Chapter 7). For example, if interactive programs (including interactive subprogram components of distributed applications) are run locally and batch programs are run remotely, then guest program count is a reasonable load metric to use for host selection for remote execution (assuming that significant local activity always

results in preemptive migration of guest programs). Thus, the requirement in Section 6.4 to use guest program count as load metric is actually not a detriment—as long as we can separate the interactive and batch workload components from each other.

Chapter 7 describes an implementation of a system in which users manually separate the interactive and batch program workload, running interactive programs locally and batch programs remotely. The performance measurements taken for host selection for this system verify our claim that guest program count provides a load metric for host selection that is comparable to processor utilization. Comparable is defined here to mean that the throughput for batch programs is the same when guest program count is used as the load metric.

At high loads, guest program count should actually out-perform processor utilization as a load metric because we must *average* the processor utilization of a machine over some period of time in order to smooth out its temporary fluctuations.[4] This is especially so if we allow guest programs to be placed on machines where local interactive activity—averaged over time—is small but nonzero. Unfortunately, this averaging implies that when a guest program terminates, the load on its machine will still reflect its previous presence for some period of time. If host selection requests occur frequently enough, then they will miss an inherently lightly loaded host candidate for this reason. This should show up especially with applications that consist of multiple programs that are executed in sequence, one immediately following the other. In contrast, guest program count immediately reflects when a machine is newly available for more work.

Of course, combinations of various load metrics can be used to improve their accuracy. At the simplest level, if both processor utilization and guest program count are available, one could create a load metric that is a weighted average of the two. If more information is available, such as a more detailed processor utilization history (for example, averages over 0.5, 1, 2, 5, and 10 seconds of time), then this can be used to construct even more accurate load metrics. However, the point of this section is not to elaborate on the many variations of load metrics possible, but to demonstrate that the load metric used for our application load balancing scheme is a reasonable one to use in general if we can separate interactive and batch programs from each other.

6.7 Who Does Load Balancing

Thus far no mention has been made of who coordinates migration operations for load balancing. The obvious candidates one might think of are the lightly loaded machines to which programs will be migrated. Assuming that applications aren't forced to coordinate migration themselves, this would require that the program manager, or some equivalent server on these machines do so instead. This, in turn, implies that these servers would have to be told that they are running subprograms of a distributed application that desires this migration service. Having a central server instead, would avoid the troubles of coordinating multiple servers to retain this information. Thus, we advocate having one global load balancing server with which all applications desiring migration service register themselves.

The manner by which the load balancer can coordinate whom to migrate when must rely on some means of determining when subprograms terminate and some means of determining the load seen by each subprogram at such times. Since we wish to find out about terminations quickly, having the load balancer poll subprograms for liveness will generate considerable network traffic if many subprograms are involved. The alternative is to request a termination notice for each subprogram from some party that is local its machine; for example, from the program managers that load them. If multicast groups are cheap to create and destroy in the operating system being used, then a new multicast group can

[4]This claim could not be experimentally verified with the system described in Chapter 7 due to limitations of the file servers available at the time that experiments were being run.

be created for each application that includes only the subprograms—or hosts, depending on what level the distributed operating system provides multicast at—of the application. This group can then be queried whenever a subprogram terminates in order to locate a suitable migration candidate, if any. If multicast groups are expensive to create then a single multicast group could be used for all applications' subprograms. If distributed applications have many subprograms or if only a single multicast group is used, then it will be necessary to apply the reply set restriction techniques described in Chapter 5 to limit the reply traffic.

There is no reason not to combine the load balancer with a general central scheduler, if one is being used in a system. In fact, there is no reason not to extend the load balancer to more general load balancing algorithms, as long as the amount of resources it takes up is limited. Since a central scheduler already knows about lightly loaded machines and how often new remotely executing programs can be expected to appear, all it needs additionally is a set of candidate programs to (possibly) migrate.

One might consider creating a multicast update group of heavily loaded hosts, similar to the light load update group of Chapter 5. Whether or not load balancing is useful for general programs is a topic that will not be addressed in this thesis. However, for the distributed applications we have been worrying about, the scheduler must already determine a set of heavily loaded machines whenever an application's subprogram dies. There is no reason not to also consider migrating these subprograms onto *any* lightly loaded machines at such times.

Note that we are implicitly justifying the overhead of migration to the system's throughput by assuming that the cost of idling resources when a distributed application is slowed is larger than the cost of migration. Unfortunately, the validity of this assumption is very application-specific and system workload-specific, depending, for example, on the cost of migrating subprograms, how many subprograms there are, what their runtimes are, and how close to fully loaded the system is.

6*8 Migrating to Newly Idle Resources

In order to illustrate some of the additional issues one must consider if the provision of general load balancing facilities is really desired, we will consider briefly how newly *idled* resources might be acquired by migrating programs onto their associated machines. If a central scheduler is being used to perform the host selections for remote execution, then it will have to interact with the program managers of each machine in the system to gather the necessary information for deciding when to migrate programs onto newly idled resources and when to simply wait for a new remote execution request to acquire them "automatically". If the system is using a decentralized scheduling design, then the relevant information must be maintained by other means.

Migration will be justified only if the cost of migration is smaller than the number of additional resources gained. Considering just CPU cycles, if we use the definitions from Section 6.4 and let T_r be the expected time until the next remote execution request, then we should migrate a program if the number of idle cycles thereby acquired is larger than the number of cycles lost on the source machine to migration:

$$fd \cdot (co \cdot T_r - c_m) > c_m.$$

This assumes that the migrated program will run for at least the time interval T_r and that the source machine is fully loaded. (Clearly the source machine had better be fully loaded to justify migrating a program off it.) Simplifying this inequality yields:

$$T_r > \frac{1 + f_d}{f_d} \cdot \frac{c_m}{c_0}. \tag{6.2}$$

In order to reach a scheduling decision, a scheduler will have to know or estimate both T_r and c_m in the above inequality. Obtaining a value for c_m implies either knowing which program to migrate ahead of time or using an average migration time value for all programs. However, there is also another factor to consider. Which program to migrate depends upon:

- the cost of migrating it, which will be determined primarily by the size of its address spaces,
- how much longer it is expected to run.

Not only do we wish to minimize the cost of migrating any particular program, we also wish to minimize the number of migrations that occur over time by picking programs that are expected to run a long time. This requires estimating the expected completion times of programs, which usually can only be done crudely by extrapolating each program's current time of existence an equal time into the future. In turn, it also implies that the system's load balancing facilities should keep track of the longest running programs in the system.

To keep track of the longest running programs, each machine must keep track of how long all programs on it have run. Each machine must then propagate information about its most suitable load balancing candidate to whomever is performing the actual scheduling decisions for load balancing. In a centralized design, this will be the central scheduler—who already receives state updates notifying it of newly idle machines. In a decentralized (query-based) design, the appropriate scheduler should be each newly idle machine itself. Upon becoming idle and deciding that a new remote execution request will not be forthcoming soon, a machine should query the system for load balancing candidates and then pick the most appropriate candidate. In either case, we will want to reduce the communication traffic involved by only receiving candidate information from the most likely candidate machines by using the techniques described in Chapter 5.

If we assume a random arrival rate for remote execution requests that is modeled by a Poisson process, then the average time between arrivals, $\bar{T}_r$, can be used for T_r in Equation 6.2. A central scheduler can easily estimate a value for the average time between remote execution requests, $\bar{T}_r$. In a decentralized design, the newly idle machines will have to determine a value to use for $\bar{T}_r$ without the benefit of past history to rely on. This will require that some set of machines in the system, for example, the members of the light load multicast group, maintain a running estimate of the remote execution frequency. Then newly idle machines could query this set in order to obtain an estimate for $\bar{T}_r$.

One might consider the alternative approach of trying to estimate lower bounds on $\bar{T}_r$ by assuming a distribution for remote execution requests and waiting sufficiently long for none to arrive. Unfortunately, while we wait to determine that $\bar{T}_r$ is long enough, we also lose much of the time during which migration would have helped make use of the idle resources in the first place!

6.9 Summary and Conclusions

In this chapter, we have considered questions of fairness and a scheme to load balance the subprograms of distributed applications. Fairness can mean many things and attempting to achieve different kinds of fairness can lead to conflicting goals, such as attempting to provide equal resource shares to all users of the system while simultaneously trying to guarantee to distributed applications that their subprograms will get equal CPU shares among themselves.

The principal thrust of this chapter has been twofold: First, we claim that the issue of fairness can largely be ignored! Nevertheless, we have described a global scheduling policy and load balancing scheme that allows us to accommodate one type of fairness policy that

we consider particularly important, namely providing approximately equal CPU shares to the subprograms of distributed applications whose runtimes are determined by the speed of their slowest subprograms.

Our claim that fairness is not an issue most of the time is based on the following observations:

- The concept of remote execution implicitly includes the assumption that there are usually many idle resources in the system available for use. As long as the system has sufficient resources to handle all remotely executing programs, fairness is not an issue since there is no competition for resources among programs. Similarly, if the system is *fully* loaded, meaning every machine is being fully utilized, then remote execution—with its attendant issues of global fairness and load balancing—is pointless since there are no extra idle resources to gain access to.
- We assume that users mainly care about fairness in the long term—averaged over multiple program executions. As long as their programs are not starved by complete exclusion from execution, they will be willing to accept short term variations in load. We assume that local activities that might exclude remotely executing programs from execution monopolize their machines through preemption, thereby avoiding the problem of exclusion. (Note that without the ability to preemptively migrate guest programs, we cannot make *any* assumptions about minimum resource shares of programs.) Thus, if users generate equivalent workloads, then disparities in fairness among users will average to zero, yielding "statistical fairness".
- Users that wish to run large, distributed applications that would bog down the system if allowed to compete equally with other applications can be dealt with by providing a guest background priority scheduling class that only receives CPU cycles when no other programs are runnable. Programs would be placed in this priority class *explicitly* by users (thereby requiring no work on the part of the operating system) and would be suitable for any applications that are not unduly sensitive to variations in the resource shares they receive.

"Statistical" fairness among users is not sufficient for distributed applications whose overall runtimes are determined by the speed of their slowest subprograms. We have shown that a global scheduling policy based on a least loaded host selection algorithm, using guest program count as its load metric, can be used to approximately equalize the guest program load on all machines (not dedicated to local activities) in the system. Preemptive migration on any significant local activities and round-robin scheduling on each machine are required. This scheduling policy assures that, as long as the current number of remotely executing programs in the system does not exceed the average number of such programs by more than a factor of two, every machine will have at most twice as many guest programs running on it than the average number throughout the system. This guarantees that all remotely executing programs will receive at worst half the CPU share that is provided to the average remotely executing program in the system. If the cost of migration is small relative to the runtime of typical programs, then migration can be used to deal with temporary load variations that exceed a factor of two by migrating programs from excessively heavily loaded machines, implying that the bound of a factor of two can almost always be achieved. Note that this scheduling policy still allows the provision of statistical fairness among users.

For the important class of distributed applications whose subprograms work on independent tasks—and hence don't need to communicate with each other—we introduced a simple migration scheme that can be used to further narrow the differences in CPU shares received by an application's subprograms: The migration scheme simply migrates subprograms from heavily loaded machines to lightly loaded machines as they become available. By starting out from a relatively equalized load distribution, this scheme is able to come fairly close to achieving the effect on application turnaround times that would be obtained if all subprograms of an application received equal CPU shares. Specifically, as long as the

system's remote execution load does not temporarily exceed its average load by more than a factor of two, the difference in turnaround times will be at most 17% different, plus the cost of migration. Subprograms will be migrated at most once more than the number of times they are preemptively migrated—which we have assumed to be infrequent. (Note that subprograms won't necessarily be migrated at all.) If the duration of a subprogram is much longer than the time required to migrate it—for example, if it runs for several minutes and takes only a few seconds to migrate—then the overhead of migration in our scheme will be negligible. Whereas improvements on this load balancing scheme might be achieved by making assumptions about the runtimes of programs, it is unclear whether the improvements would be worth the effort given that they would very likely require application-specific assumptions.

If the interactive and batch components of the system workload can be separated from each other, then guest program count provides a slightly better selection metric than does processor utilization for remote execution. This is because processor utilization has problems with latency: In order to smooth out instantaneous fluctuations in load we must average processor utilization over some period of time. But this causes machines on which a guest program has just terminated to be mistakenly seen as still loaded for a brief period after they are really available.

The final topic we have covered in this chapter is a discussion of several issues pertaining to load balancing facilities in general, such as how to obtain information about the potential migration candidates in the system and under what conditions migration of a program is worth the overhead. The intent has been mainly to illustrate the mechanical and informational needs that must be addressed in the provision of load balancing rather to provide a guide to its applicability. However, in line with our arguments at the beginning of the chapter, we feel that load balancing facilities will only be necessary if the sorts of distributed applications we have described become prevalent or if a system runs "skewed" workloads, such as several large distributed applications late at night, where the payoff of load balancing clearly outweighs the overhead incurred by migration.

Chapter 7

An Implementation

7.1 Introduction to the V-System

This chapter describes an implementation of many of the concepts discussed in the previous chapters. Since the general concepts and algorithms have already been covered, the material presented here will emphasize notable details of the implementation that are specific to the system and environment actually used, in addition to a quantitative assessment of the facilities built.

The V-System[Che84,Gro86], a message-based distributed operating system designed primarily for high-performance workstations connected by local networks, was used as the testbed upon which this implementation was built. Aside from being an existence proof, the implementation has provided us with a quantitative assessment of many of the concepts and requirements needed for preemptable remote execution facilities.[1] Unfortunately, the actual testbed system only contains about 40 machines and the workload placed on the system is not yet very large. This has prevented us from evaluating issues of scalability and the associated workload assumptions directly; we have only been able to extrapolate from our current, relatively light workload. This does not, however, prevent assessment of the load limits that can be placed on scheduling facilities in terms of such things as the number of host selection requests, for example.

The V-System permits a workstation to be treated as a multi-function component of a distributed system, rather than solely as an intelligent terminal or personal computer. This system has been built by the Distributed Systems Group at Stanford University and is used primarily as a testbed for research into distributed systems issues. Ultimately, it is intended to provide a general-purpose program execution environment, similar to some degree to UNIX, in which distributed programs interact with each other to produce an integrated system.

The V-System consists of a distributed kernel and a distributed collection of server processes. A functionally identical copy of the kernel resides on each host in the computer system that is running as a native V host. This kernel provides address spaces, processes that run within these address spaces, and network-transparent interprocess communication in the form of synchronous message passing. Both unicast and multicast message communications between processes and groups of processes are provided. All other services provided by the system are implemented by processes running outside the kernel. Machine-relative services include exception handing, program loading and execution, terminal handling and graphics, and command interpretation. System-wide services provide

[1] The performance figures presented in this chapter are for SUN-2 workstations with a 10 MHz Motorola 68010 processor and 2 Mbytes of local memory. Workstations are connected by a 10 Mbit Ethernet local area network. All measurements were taken in the fall of 1985 and the spring of 1986 using an experimental version of release 5.1 of the V-System software.

file storage, authentication, configuration monitoring, and various specialized services such as access to printers.

7.1.1 The Distributed Kernel

The principal facilities provided by the V-Kernel are processes and communication between processes via messages. A process is identified by a globally unique *process identifier*, or *pid*, which is used to specify senders and receivers of messages. Messages may optionally contain a capability to access a *segment* of the sender's address space, which can then be read or written using interprocess copy operations. Messages themselves are fixed-length 32-byte entities, but segments may be of arbitrary length. Access to the segment is rescinded when the receiver of a message replies to it.

The V-Kernel allows multiple light-weight processes to execute within each address space, which is called a *team*. Teams reside within *logical hosts*, which represent virtual physical hosts. Processes are referenced by identifiers that are constructed as *(logical-host-id, local-process-id)* pairs. Logical host-ids provide a means of indirection that allows the kernel on each physical host to map them to the appropriate physical network address to use with interprocess communication. These mappings are cached in the kernel, with broadcast being used to resolve cache misses. Although this indirection is exploited to make logical hosts the unit of migration with the V-System, it is necessary in any case since Ethernet network addresses are 48 bits long, whereas process-ids are 32 bits long. Each physical host can support several logical hosts, allowing logical hosts to be used to contain individual applications.

In an environment of many cooperating processes on different machines there are many logical *groups* of processes that provide equivalent services. Examples include file servers and a group of processes executing a distributed, parallel computation. The V IPC mechanism has been extended to allow messages to be sent to such a group[CZ85]. Access to *well-known* process groups, such as the set of all file servers, is obtained through predefined group-ids. Group-ids are identical in syntax and similar in semantics to process-ids. However, unlike regular unicast IPC, multicast operations are "best effort" only; the V-Kernel does not guarantee that multicast messages will be delivered to every intended receiver, nor that all replies will be received.

In order to provide efficient access to groups of processes that are known to be local to a particular machine, *local* process groups are provided. These are constructed by concatenating a logical host-id with a local group-id rather than with a local process-id and are thus actually relative to logical hosts. However, their scope covers an entire physical host, allowing processes outside a logical host to join its local process groups. This becomes most useful when the concept of *well-known local process groups* is introduced, which allows machine-relative access to groups of processes in a location-independent manner. References are relative to the location of a logical host rather than a particular physical machine.

The kernel is structured as three major components: IPC, a kernel server and a device server.[2] The IPC provides the basic glue for connecting the different system components and is implemented with conventional "system call" traps. The kernel server is a process within the kernel that provides process and memory management operations as a server accessed by the IPC. The device server is a pseudo-process, currently implemented by the kernel server, that provides access to devices through messages. The device server uses the V I/O protocol for input and output with these devices. This protocol, described in [Che86], is used by all servers in the V-System that support I/O-like activity, thereby providing a single uniform I/O interface to all clients.

[2]The version of the kernel described here provides multiple virtual address spaces, but without demand-paging support.

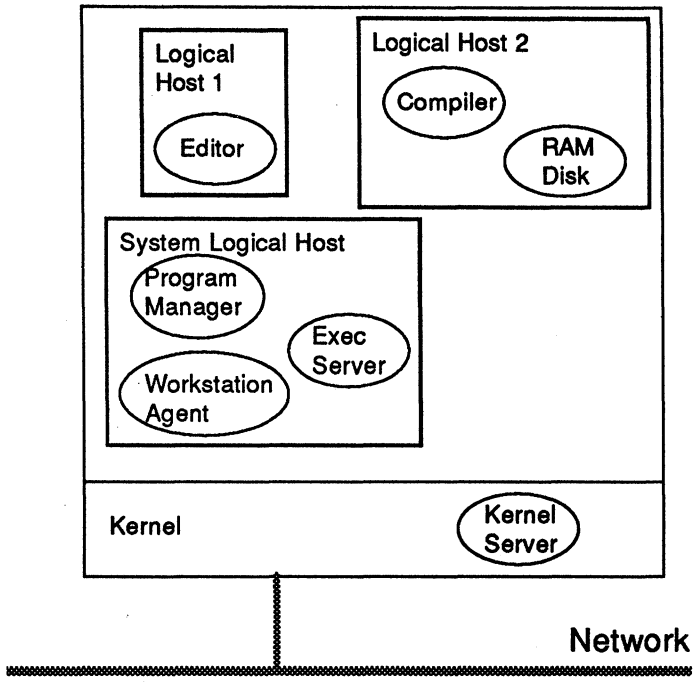

Figure 7.1: Example of a V-System workstation.

Although the kernel and device server of each machine are accessed via messages, in order to avoid buffering problems within the kernel, most operations requested of these servers that require access to memory segments may be invoked only locally. (Thus, the servers can access the memory segments directly instead of having to create separate internal buffers to store data obtained from remote processes.) [3] This implies that I/O operations to the device server are, for the most part, constrained to be performed by local processes.

Security in the V-System is based on the concept of *user-ids*. Each authorized user within a V domain is assigned a unique user-id, and each V process bears exactly one user-id. A process runs with the privileges associated with its user, and that user is considered responsible for its actions. User-ids are maintained by the V-Kernel and all messages sent by processes include their associated user-id as a capability. This mechanism assumes that the sanctity of the distributed kernel cannot be violated: Messages sent using the kernel on one machine are assumed tamper-proof before they reach the kernel on another, and it is assumed that rogue versions of the kernel cannot be substituted for the authorized version.

7.1.2 Servers

There are two kinds of servers in the V-System: machine-relative servers and global servers. Machine-relative servers can be instantiated on each machine in the system (but don't have to be) and are mainly dedicated to providing local operating systems functions for the machine on which they are running. Global servers manage objects/services that are

[3] Only segment-based operations that can be immediately finished, and therefore handled using a single kernel buffer, are supported remotely. Fortunately, this includes most kernel server operations.

system-wide in their scope, such as network file systems or printer services. Figure 7.1 shows a snapshot of the team and logical host structure as it might appear on a typical V-System workstation.

The Program Manager

Aside from the kernel (and associated device) server already described, the *program manager* is the only machine-relative server that must be present on every native V host in the system. The program manager is essentially the manager of its physical host, and much of its function is similar to that of the *Butler* and *Banker* processes proposed by Dannenberg[Dan82]. It loads, executes, and monitors all teams other than the *first team*, which is the team in which most of the local operating system servers reside, including itself.[4] It acts as a local agent for remote execution requests and implements a policy database concerning acceptance of execution requests (both local and remote).

The program manager also controls access to the machine's resources, principally by implementing a multi-level round-robin scheduling scheme for CPU access. In particular, local programs receive service before guest programs. Access to memory resources is currently left uncontrolled, although the kernel operations that control the size of the valid virtual address space of a team could easily be changed so that only the program manager could invoke them on behalf of a client team.

Exception conditions are also handled by the program manager: Hardware exception traps are converted to messages that are sent to appropriately registered exception handlers (that may reside on remote machines). If nobody has registered themselves for an exception from a particular process, then the program manager loads a debugger program and forwards the exception message to it. Registration of exception handlers is done by registering for exceptions from a specified process and all its descendants. Thus, exception handlers can be hierarchically stacked; however the hierarchy does not extend beyond the scope of a single team since the program manager is the ancestor of all teams on a machine. This point will be important for migrating programs without causing exception handling information to be spread over multiple machines.

Finally, the program manager acts as the representative of its machine to the rest of the V-System by answering all state information queries and optionally updating a central scheduling server. Thus, the program manager also represents the local agent of both the centralized and the decentralized global scheduling designs we have implemented for the system.

Although the program manager represents a monolithic entity to its clients that implements a variety of services, it is in fact implemented as a collection of cooperating processes that all reside in a single address space.[5] Thus, the benefits of modularity and concurrent threads of control are not lost in this design. However, migration of machine-relative server state information is greatly simplified by coalescing all relevant data structures into a single address space where they can be locked and extracted in a single operation.

Other Machine-relative Servers

Although the program manager is the only machine-relative server that *must* be present on every V host, most machines typically also run a workstation agent and an *exec* server. These servers mediate the interaction of human users with a machine.

[4]Typically teams correspond to programs, although programs may consist of more than one team. Teams are always loaded with an executable binary image of a program.

[5]In fact, much of the documentation of the V-System refers to the program manager in terms of the two separate server modules that comprise it, namely the team server and the exception server.

Workstation agents are a generic class of server in the V-System, the particular instantiation used being determined by the I/O devices available on a machine. A workstation agent mediates between the I/O hardware, the human user, and the other programs—including servers—in the system. It is called upon by programs or other servers to receive input from devices controlled by the user and to display output to the character/graphics displays seen by the user. Workstation agents send request messages to the kernel device server to read from the user's keyboard and mouse and write to the user's output screen, typically a graphics frame buffer. They also provide multiple I/O streams in the form of "virtual terminals" that can each be viewed through multiple windows. Thus, they multiplex a machine's I/O devices among all the programs that use them.

All interaction by programs with the user is mediated through the workstation agent of each machine, so that programs themselves only employ the message-based V I/O protocol.

The other machine-relative server typically present is the *exec* server. It corresponds to the UNIX shell or the Tops-20[6] Exec in that it provides multiple instances of a user-level process that parses a user's commands and acts as an agent for executing them. As such, it is the principal client of the program manager and the workstation agent. The exec server also maintains various "environment variable" bindings, such as home file server and current working directory, that are passed in to programs as part of their initialization.

Global Servers

Not all V-System servers run on each machine: First, some servers run only on dedicated machines; for example, printer servers only run on machines dedicated to spooling files and driving attached printers. Second, there are two crucial system-wide services provided: network file service and authentication. The V-System is oriented towards supporting diskless workstations. Thus, all file servers are global in scope and typically run on dedicated server machines. However, nothing prevents any machine from running a local file server since they are treated as just another user application.

The *authentication server* maintains a database of information about each user, including (at minimum) login name, personal name, encrypted password, and user-id. It supports simple queries on this database, which is keyed by user-id and login name, and will also set the user-id of a requesting process if the correct password for that user-id is presented in the request.

7.1.3 Naming

Conceptually, the V-System's naming facility is a system-wide global directory providing reference by high-level name (i.e. user-assigned character string name, such as file name, user account name, etc.) to objects implemented by multiple object managers, or servers[CM85]. The global directory contains a (name, object) tuple for each binding of global name to object. Each client may also have its own directory of bindings from local names (or aliases) to global names.

The design is decentralized and the global directory is distributed across the object managers such that each object manager stores and maintains that portion of the directory corresponding to the objects it implements. Each client (i.e. each team) maintains a cache of bindings from name to object manager that are manipulated by means of a suite of library routines, as illustrated in Figure 7.2. When a client invokes an operation using a high-level object name, the client checks its cache for an entry that maps the name to an object manager. If a cache entry for the name is found, the operation and name are then forwarded to the object manager indicated by the cache entry. Otherwise, a

[6]Tops-20 is a trademark of Digital Equipment Corporation.

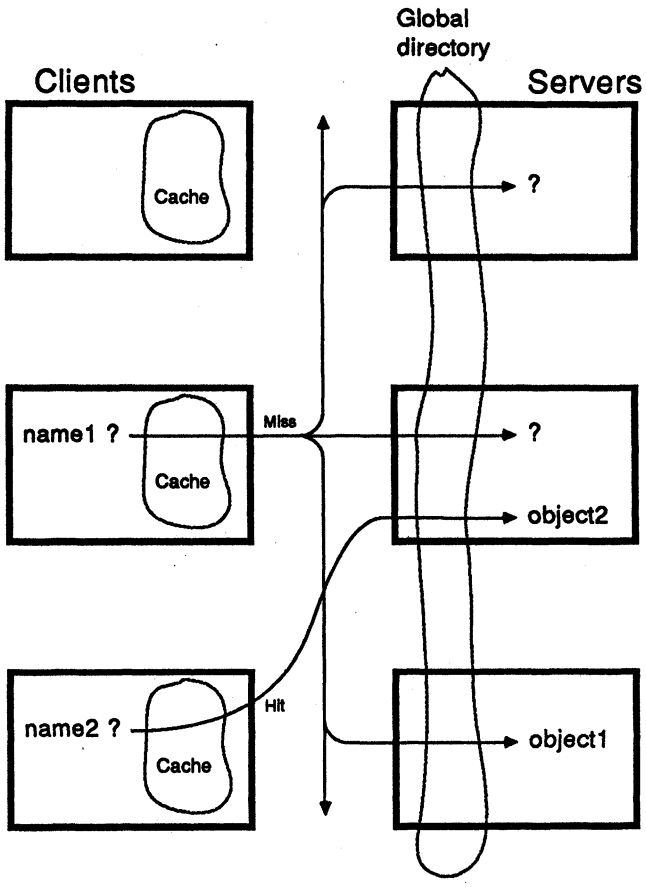

Figure 7.2: Decentralized naming using multicast.

query is multicast to all object managers to determine the correct object manager for the named object. If an object manager responds, a cache entry is created and the processing of the request proceeds as before, with the operation being forwarded to the responding object manager. Otherwise, the specified object name is assumed to be invalid and an error indication is returned to the client. The naming cache used by each program is initialized to contain various pre-defined entries when the program is first instantiated by the program manager. These pre-defined bindings are passed in to the program as part of its "environment variable" arguments.

7.2 Remote Execution

A V program is invoked at the command interpreter level by typing:

program <arguments> [Q <machine-name>]

If no machine name is specified, then the local machine is assumed as the default. Using the meta-machine name any:

<program> <arguments> Q any

executes the specified program at the "least loaded" machine on the network. A standard library routine provides a similar facility that can be directly invoked by arbitrary programs. Any program can be executed remotely providing that it does not require low-level access to the hardware devices of the machine from which it originated. Hardware devices include disks, frame buffers, network interfaces, and serial I/O lines. Specifying the special machine name `local` forces a program to be kept local and never migrated.

Remotely invoked programs are instantiated in new logical hosts so that they can be individually migrated as needed. In order to allow grouping of multiple teams of a single application that should stay together (e.g. a compiler and its private RAM disk server for temporary file storage), locally invoked programs are placed in the same logical hosts as their invoker unless explicitly requested otherwise. This provides a simple semantics for grouping programs into logical hosts, but implies that new commands to be executed locally must be explicitly requested to be placed in a new logical host by their invoking procedure. Preventing programs from being migratable is implemented by placing them into the system logical host, which can never be migrated.

A suite of programs and library functions are provided for querying and managing program execution on a particular workstation as well as all workstations in the system. Facilities for terminating, suspending and debugging programs work independent of whether the program is executing locally or remotely.

It is often feasible for a user to use his workstation simultaneously with its use as a computation server. Because of priority scheduling for locally invoked programs, a text-editing user need not notice the presence of background jobs providing they are not contending for memory with locally executing programs.

7.2.1 Implementation

Initiating local execution of a program involves sending a request to the local program manager to create a new address space (optionally on a new logical host) and load a specified program image file into this address space. The program manager uses the kernel server to set up the address space and create an initial process that is awaiting reply from its creator. The program manager then turns over control of the newly created process to the requester by forwarding the newly created process to it. The requester initializes the new program space with program arguments, default I/O, and various "environment

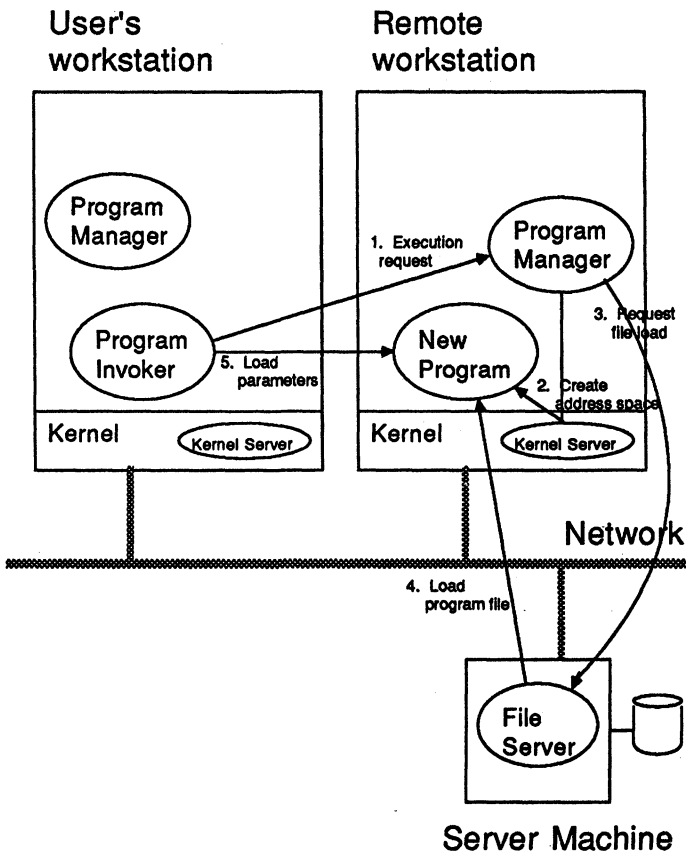

Figure 7.3: Communication paths for program creation in the V-System.

variables", including a name cache for commonly used global names. Finally, it starts the program in execution by replying to its initial process. The communication paths between programs and servers are illustrated in Figure 7.3.

A program is executed on another workstation by addressing the program creation request to the program manager on the other workstation. The appropriate program manager is selected using the global scheduling facilities described in Section 7.4.2.

Beyond selection of a program manager, remote program execution appears the same as local program execution because programs are provided with a network-transparent execution environment, assuming they do not directly access hardware devices. In particular:

- The program address space is initialized the same as when the program is executed locally. For example, arguments and environment variables are passed in the same manner.
- All references by the program outside its address space are performed using network-transparent IPC primitives and globally unique identifiers, with the exceptions of the host-specific kernel server and program manager. For example, the V I/O protocol implements byte streams by means of messages directed to `(server-pid, instance-id)` pairs. The `server-pid` part is a (global) IPC identifier and the `instance-id` is an integer that is unique relative to the server specified by the `server-pid`.
- High-level naming is implemented as described earlier, implying that there is no local name server in each machine to initialize and deal with.
- Access to the kernel server and program manager of the workstation on which a program is running is obtained through well-known local process groups, which in this case contain only a single process. Each machine-relative server can be accessed by constructing a process group-id consisting of the program's logical-host-id concatenated with the well-known local group-id for the server. Thus, the kernel server and program manager must join a well-known local process group for each logical host running on a machine. In this manner machine-relative servers can be referenced in a machine-independent manner.
- We assume that remotely executed programs do not directly access a device server. The limitation on device access has not been a problem in V since most programs obtain adequate access to physical devices through server processes that remain co-resident with the devices that they manage. In particular, programs perform all normal user interaction via a workstation agent that remains co-resident with the I/O devices it manages[LN84,Now85].

7.2.2 Evaluation

Performance Measurements

The time cost of remotely executing a program can be split into three parts:

- Selecting a machine to use,
- setting up and later destroying a new execution environment for it,
- loading the program's executable binary image into its address space from a file server.

Of these, the cost of loading the executable binary image considerably dominates the other two. Since workstations are typically run diskless, program files are loaded from network file servers, making the cost of program loading independent of whether a program is executed locally or remotely. This cost is a function of the size of the program to load

Program	Size (Kbytes)
build	65322
cc68	38892
preprocessor	34524
parser	141894
optimizer	29110
assembler	74214
linking loader	44026
tex	Y64SJY2
metafont	564032
ditt	49UUU
grep	24736
cp	28998
sed	32616
sort	28194
ci	S25U2
co	83114
res	81718
rcsdiff	54786
rlog	64154

Table 7.1: Memory usage of various V-System programs.

and is typically 330 milliseconds per 100 Kbytes of program from an unloaded file server. Table 7.1 lists the sizes of various commonly used programs.[7] As can be inferred, program load times are on the order of a fraction of a second to several seconds in length.

In contrast, the time required to perform the other two steps is measured in tens of milliseconds. The time to select a remote host is typically about 13 milliseconds in our system of 40 machines and is discussed in more detail in Section 7.4.2. Setting up a new execution environment requires about 15 milliseconds plus delays due to queueing and interspersal of other tasks by the various servers.

The only "non-use" cost imposed by supporting remote execution facilities is an extra overhead of about 100 microseconds imposed on every kernel server and program manager operation since these are now referenced indirectly via process group-ids rather than their own process-ids.

There is no space cost in the kernel server and program manager attributable to remote execution except for supporting global scheduling facilities since the kernel provides network-transparent operation and the program manager uses the kernel primitives. Supporting global scheduling facilities adds about 1 Kbyte, or roughly 5%, to the size of the program manager and is discussed in more detail in Section 7.4.2.

Usage Observations

When remote program execution initially became available, it was necessary to specify the exact machine on which to execute the program. In this form there was limited use of the facility. Subsequently the "@ any" facility, allowing the user to specify the "least loaded host", was added. With this change use increased significantly. In general, *simple* invocation procedures proved to be important prerequisites for use of the facilities, whether at the command interpreter level or by the library routines provided for program-invoked remote execution.

Most of the use for remote execution has been for non-interactive "batch" jobs with

[7] Note that these sizes represent only the code, initialized data, and symbol table parts of each program.

non-trivial run times, such as compilation or text formatting. Since the default for invocation is to execute programs locally, interactive programs and "short" programs such as directory listing are typically executed locally. Users typically run one or more interactive programs locally, of which usually only one is actively running at any time, and execute one or more "batch" jobs remotely in parallel. As a consequence little experience is available with running *all* programs "`@ any`". The reason why "`@ any`" has *not* been chosen as the default invocation form is due mostly to historical reasons centering around the initial absence of migration facilities and because it has not been a problem. This approach also conveniently separates the interactive and batch components of the system's workload, allowing us to eventually experiment with the load balancing scheme for distributed applications described in Chapter 6.

An area in which very positive usage experience has been gained is that of network graphics. When interactive applications must execute remotely from the graphics devices they deal with, the forms of interaction available to them must conform to the limits imposed by network IPC, as discussed in Chapter 3. Work done by Lantz and Nowicki has provided network graphics protocols together with appropriate graphics "front-end" servers that has allowed all our interactive applications to date to run remotely in a performance-transparent manner[Now85,LN84].

7.3 Preemption and Migration

A program can be caused to migrate by invoking:

`migrateprog [-n] [-h <machine-name>] [<program>]`

to remove the specified program from the workstation. If no other host can be found for the program, the program is not removed unless the "-n" flag is present, in which case it is simply destroyed. If no program is specified, `migrateprog` removes all remotely executed programs. The "-h" flag allows migration to a specified machine.

Both users and programs are able to invoke the migration program, subject to proper authorization privileges. Anyone with the same user-id as a program may migrate it. (Thus, programs can ask to have themselves migrated.) Local users of a machine preempt guest programs by asking the machine's program manager, which is a privileged server, to invoke migration for them.

7.3.1 Implementation

A program may create sub-programs, all of which typically execute within a single logical host. Migration of a program is actually migration of the logical host containing the program. Thus, typically, all (local) sub-programs of a program are migrated when the program is migrated. For example, our C compiler creates a private RAM disk storage server to store temporary files in memory. This subprogram should be migrated with the rest of the compiler. This approach is also used to ensure that debuggers can be migrated with the programs they are debugging.

The general procedure to migrate a logical host is discussed in Chapter 4. Recapitulating the steps here:

1. Locate another workstation that is willing and able to accommodate the logical host to be migrated.
2. Initialize the new machine to accept the logical host.
3. Pre-copy the state of the logical host.

4. Freeze the logical host and complete the copy of its state.
5. Unfreeze the new copy, delete the old copy, and rebind references.

In this subsection we will only discuss the details of migration of a logical host itself. Migration of its associated machine-relative server state will be described in the next subsection. Furthermore, since our implementation follows the general one discussed in Chapter 4, we will only discuss interesting V-specific implementation details here.

The first step of migration is accomplished by the same mechanisms employed when the program was executed remotely in the first place, which are discussed in Section 7.4.2.

Initialization of the new host and pre-copying the state are done as described in Chapter 4. A notable detail is that the pre-copy operation is executed at the same priority as all other programs on the originating host. We have not found it necessary for the workload we run to force it to a higher priority in order to prevent interference with its progress by other programs. Also, since a logical host may contain multiple teams, each of which may be changing its address space size, care must be taken to properly handle all changes that might occur. This may involve additional interactions with the new host's program manager to create or delete teams or to change their virtual address space sizes.

There are two notable features about how we complete the copy of a migrating logical host: First, by having all kernel operations except the basic IPC calls themselves be implemented by a kernel server process that is reached via the normal IPC primitives we are able to forward deferred kernel operations on a migrating logical host almost trivially. Second, since the V-System supports unreliable multicast communications, our implementation is forced to queue all incoming multicast messages (and their replies) for a frozen logical host at their destination instead of being able to rely on their retransmission. This, in turn, implies that care must be taken to avoid having all the reply messages of a multicast query message take up all of a machine's message buffers, thereby blocking out all other communications.

Finally, references to the processes of a logical host are rebound as follows: The only way to refer to a process in V is to use its globally unique process identifier. As defined earlier, a process identifier is bound to a logical host, which is in turn bound to a physical host via a cache of mappings in each kernel. Rebinding a logical host to a different physical host effectively rebinds the identifiers for all processes on that logical host. When a reference to a process fails to get a response after a small number of retransmissions, the cache entry for the associated logical host is invalidated and the reference is broadcast. A correct cache entry is derived from the response. The cache is also updated based on incoming requests. Thus, when a logical host is migrated, these mechanisms automatically update the logical host cache, thereby rebinding references to the associated process identifiers.

7.3.2 Residual Host Dependencies

The only machine-relative servers in the V-System that contain state that must be migrated along with a migrating logical host are the kernel server and program manager. V is oriented towards diskless workstations and all file servers are global network file servers. This has allowed us to avoid several of the difficult issues described in Chapter 3 for dealing with machine-relative servers.

The kernel server is dealt with by means of explicit *extract* and *install* operations that take a description of a logical host's kernel state and allow it to be passed as uninterpreted bytes by the user-level migration coordinator (the `migrateprog` program). Since the amount of state to transfer is small, typically less than one Kbyte, performance is not an issue. Thus, migration did not have to be implemented as a monolithic kernel operation, although the kernels on the two machines involved must still be careful to implement the transfer of control and associated kernel server operations correctly. As mentioned earlier, kernel server request messages for the migrating logical host are forwarded as normal

(duplicate suppressed) messages between kernels when the old copy of the logical host is deleted.

The program manager is handled similarly, namely by using query/install operations. Since the program manager on the new machine is the local agent for migration there, there is no need for an explicit install operation. The amount of state information to migrate is again small, so that copy performance is not an issue. The problem of correctly queueing and forwarding server requests for a migrating logical host was fortuitously solved by being able to structure all relevant operations to be idempotent. Thus, duplicate suppression did not have to be performed and the program manager on the new machine could simply join the relevant well-known local process group for the migrating logical host while the program manager on the original machine was still a member of it (with respect to the old copy of the logical host). It should also be noted that in practice duplicates only occur when a remote request message to the original program manager is lost in the network *several* times so that it is retransmitted as a broadcast packet.

7.3.3 Evaluation

Performance Measurements

The time cost of migration is similar to that of remote execution:

- A host must be selected,
- a new copy of the logical host's state in the kernel server and program manager must be made and the old one deleted,
- the logical host's address spaces must be copied.

Again, the cost of copying the address spaces between machines dominates the cost of the other two. The cost of memory-to-memory transfer in our system is about 300 milliseconds per 100 Kbytes on an unloaded network (see below). Since pre-copying involves "fragmented" copies after the initial bulk copy and involves repeated transfer of some memory pages, the times required to copy an address space are somewhat longer than the time to simply copy an address space once as a single contiguous segment. Also, since the size of the address space may change during the pre-copy operation, additional delays are incurred to continually check address space sizes and to coordinate resizing the address spaces being copied into if necessary.

Bulk memory copies between machines can also cause flow control and congestion problems if the communication protocol employed is not careful to avoid packet overloads at network interfaces or retransmits entire memory segments instead of just lost packets. Whereas program loading generates network traffic at the speed at which a file server can read data off its disks and then place it on the network, memory-to-memory copies without flow control can drive a fast network at the maximum rate at which machines can process network traffic. If the receiving machine is slower for some reason, or receives additional messages sent to it from third parties, then a bulk memory copy operation will end up frequently dropping one or more messages. If retransmission involves resending most of the original data segment, then this retransmission problem can become persistent, in addition to significantly increasing transmission times.[8]

Measurements of various application programs indicate that usually two pre-copy iterations are sufficient—one initial copy of an address space and one copy of subsequently modified pages. The resulting amount of address space that must be copied, on average, while a program is frozen is between 5 and 70 Kbytes in size, implying program suspension times between 15 and 210 milliseconds, in addition to the time needed to copy the kernel

[8]The implementation of the V-System used for the measurements presented in this thesis suffers, in part, from the problems just described.

Time interval (secs)	0.2	1	3
build	21.6	30.6	39.0
cc68	10.8	13.4	14.4
preprocessor	25.0	38.4	46.0
parser	45.0	60.4	64.6
optimizer	19.6	23.6	23.6
assembler	20.2	27.0	35.6
linking loader	20.4	37.8	70.0
tex	67.6	109.8	144.0
metafont	54.6	82.8	105.4
diff	10.0	10.0	10.0
grep	9.6	10.0	10.0
cp	14.4	19.6	22.0
sed	22.0	22.0	22.0
sort	6.6	13.6	24.0
ci	10.8	11.2	12.0
co	11.4	12.2	14.0
rcs	8.0	8.0	8.0
rcsdiff	8.0	8.6	9.0
rlog	7.6	9.2	16.0

Table 7.2: Dirty page generation rates (in Kbytes/time).

server and program manager state. Table 7.2 shows the average rates at which dirty pages are generated by various programs.[9] Unfortunately, no list processing-oriented programs were available for measurement.

The other costs of migration pale in comparison. The time for host selection is about 13 milliseconds, as mentioned before. The cost of creating a new copy of a logical host and copying its machine-relative server state is about 80 milliseconds plus the queueing delay of using the destination machine's program manager as a local agent for various operations.[10]

The execution time overhead of migration on the rest of the system is small:

- The mechanism for binding logical hosts to network addresses is needed anyway to map 32 bit process-ids to 48 bits Ethernet addresses. Thus, no extra time cost is incurred for the ability to rebind logical hosts to different physical machines. The actual cost of additional broadcast retransmissions to actually rebind a logical host's location is only incurred when a logical host is migrated. The overhead of discarding broadcast messages at each machine is about 0.4 milliseconds, which incurs a negligible load under all but very extreme migration frequencies.
- The mechanism for rebinding references to machine-relative servers, namely well-known local process groups, is already needed for remote execution and adds about 100 microseconds to the cost of every kernel server and program manager operation, as described earlier.
- 13 microseconds are added to several kernel operations to test whether a process (as part of a logical host) is frozen.

[9]These numbers depend, in part, on the load at the file servers being used. The measurements presented were taken at unsociable hours when the load was fairly low. Normal daytime usage would result in lower rates due to longer waits on the file servers. Also, the rates measured for `build` and `cc68` exclude the time intervals during which these programs wait for subprograms to complete and thus have no pages modified.

[10]The production version of the V-System also runs `migrateprog` as a program that is subject to the system's round-robin scheduling policies.

Program		Size (bytes)	Migration time (secs)	Freeze time (secs)	Aggregate transfer rate (Kbytes/sec)
build	+ cc68 + preprocessor	308884	1.40	0.19	221
build	+ cc68 + parser	407192	2.20	0.28	185
tex		761368	3.16	0.18	241
metafont		570484	2.66	0.26	214
diff		60856	0.54	0.16	113
grep		36776	0.43	0.13	85
cp		56968	0.39	0.11	146
sed		75316	0.44	0.12	171
sort		721636	2.58	0.13	280
ci		124360	0.82	0.16	152
co		119576	0.60	0.13	199
rcs		129996	0.76	0.16	171
rcsdiff		302444	1.63	0.26	186
rlog		100488	0.63	0.14	160

Table 7.3: Migration times for various programs.

Table 7.3 shows the migration times and freeze times of various programs commonly executed remotely in our system. As can be seen from the aggregate transfer rate measure, the effective speed of migration is only about half to two-thirds the speed of 333 Kbytes/sec at which bulk memory copies can be done *on an unloaded network*. This is due mainly to the network protocol problems described earlier and to the fixed overhead incurred in addition to the cost of just copying state between machines.

The space cost of migration facilities can be divided into three parts:

- Several new kernel operations and related support code added about 9 Kbytes, or about 10%, to the size of the kernel.[11]
- The addition of a migration module to the program manager increased its size by 2.3 Kbytes, or about 13%.
- The bulk of the code needed to implement migration resides in the `migrateprog` program, which is about 44 Kbytes in size. Fortunately, this program is resident only during the actual migration of a program.

Usage Observations

To date, very limited experience is available with preemption and migration. Since our workstations are typically 80% idle and there are frequently several workstations without local users available, preemption has not been needed very often. The ability to preempt has so far proven most useful for allowing very long running simulation programs to run on idle workstations in the system and then migrate elsewhere when their users login to use them. Similarly, various system servers, such as the authentication server, are run on idle workstations rather than dedicated server machines and are migrated as necessary. We expect that preemption will receive greater use as more applications are ported to the V-System.

One important negative observation that has been made is that migration facilities are fragile. Transparent migration essentially requires being able to freeze the state of a logical

[11]The size of the kernel is measured *without* including descriptors for processes and teams since the number of these allocated depends on the particular memory configuration of the machine being run on.

host at any time, regardless of what state it is in. More specifically, we must be able to freeze, extract, and later install the *kernel state* of a logical host irrespective of what state the Jkernel^ is in, implying that the corresponding kernel operations must be intimately familiar with the internal workings of much of the kernel's implementation. In a research environment, where the kernel is frequently modified, migration facilities are easy to break unless all people working on the kernel are familiar enough with the migration facilities to keep them synchronized with all kernel modifications. Unfortunately, it is not clear how well this problem can actually be dealt with.

7.4 Global Scheduling

Both a central scheduling server and a decentralized scheduling scheme have been implemented for the V-System. The latter is the one currently in use. In both cases, status information is maintained by the program manager on each machine. This information can be obtained by querying a program manager for it. The program manager can also be placed in a mode where it updates a designated central scheduler every time a "significant" change in state information occurs.

The information stored by each program manager includes the following:

- Processor utilization, averaged over the last 10 seconds.
- Processor speed weighting factor. Since different processors have different capabilities, this number must be multiplied with the processor utilization to obtain a normalized indication of how many CPU cycles are actually available on a given machine. Currently our system includes three different processor types: Motorola 68010s, Motorola 68020s, and MicroVax-II processors.
- Number of teams currently residing on the machine.
- Number of guest teams currently residing on the machine.
- Amount of free memory available.
- Machine hardware information. This includes processor family (e.g. Motorola 680X0), processor type (e.g. Motorola 68010), and machine type (e.g. SUN-2).

Scheduling is currently done using the normalized processor utilization values. Specific processor and machine types can be requested, in which case all other machine types will be excluded from consideration by the scheduling facilities.

7.4.1 A Central Scheduler

In order to evaluate basic scheduling costs a very simple central scheduling design was implemented. A list of machines, ranked by load, is maintained by a scheduling server. The size of the list is bounded by multicasting a cutoff load value—namely, the normalized processor utilization—to the multicast group of all program managers so that only a subset of them will reply.

Figure 7.4 shows response times for host selection as a function of the number of machines, n, updating the scheduler, assuming that machines update the central server every / seconds. In order to control the loads and parameters of the program managers, a parallel set of simulated program managers was created to generate the data for the graphs and tables presented in this section. The experiments were run entirely on SUN-2 machines and host selections placed no memory restrictions, so that the first list element is always picked by the scheduler. In order to force pessimistic processing times for update messages, the loads presented by the program managers were uniformly distributed and randomly

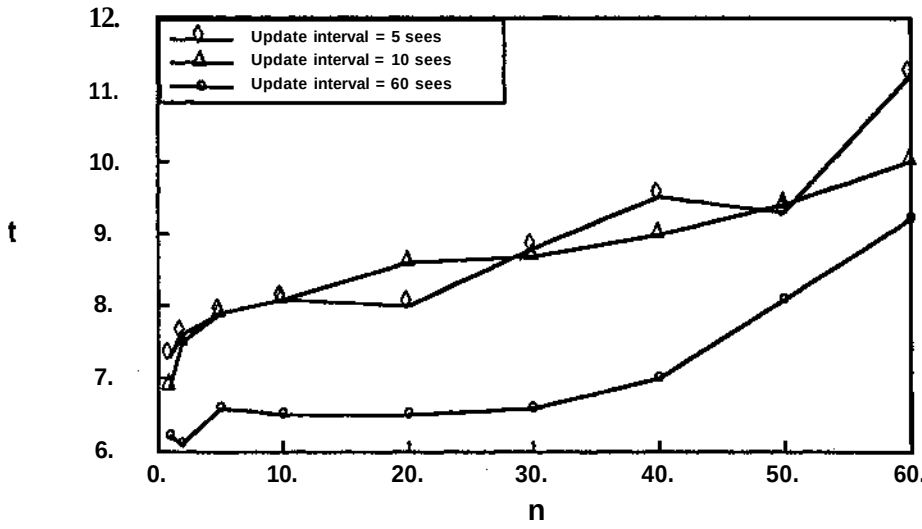

Figure 7.4: Central server host selection response times as a function of the number of updaters (in milliseconds).

changed every update interval. An update rate of once every 5 seconds illustrates what we feel is a reasonable upper bound on the update traffic one would obtain in any system. Since our system is currently 80% idle most of the time, using the actual update rates obtained from program managers that only update on significant changes in state would yield almost no update traffic at all, thus corresponding more to the curve for updates once every 60 seconds. One can argue that this will also hold in the future for the set of machines that reside on the "least loaded host" list in the server.

7.4*2 Decentralized Scheduling

The decentralized scheduling scheme employs library routines within each team that query a well-known "light-load" multicast process group containing the program managers of lightly loaded machines. The size of the light-load group is controlled by having the enquirer check how many replies are received by the machine and sending an adjusted cutoff load message to the well-known process group of all program managers if necessary. Since our system only contains 40 machines the "light-load" group currently simply contains all machines in the system for normal usage.

On most of our workstations multicast is implemented in software on top of the broadcast facilities inherent to an Ethernet. Some workstations have network interface hardware that provides multicast support directly. Measurements by Cheriton, Stumm, and Berc[CSB85] indicate that it takes about 0.4 milliseconds for the kernel to discard an unwanted multicast packet in the former case. In those machines with added hardware support this overhead drops to zero.

The overhead incurred to a program manager for replying to a query message has been measured to be 4 milliseconds.

Program managers reply to query messages after a random delay period in order to avoid flooding the sender with too many replies at once. The random reply delay is determined using a random number between 0 and 1 that is weighted by the normalized processor utilization of the replier's machine and then multiplied by a reply delay interval value that represents the time interval over which replies are to be generated. Unfor-

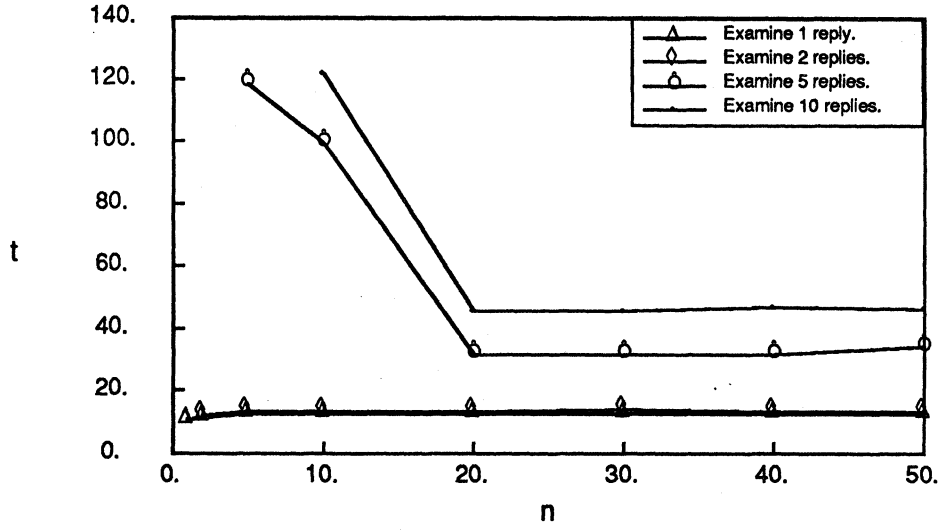

Figure 7.5: Decentralized scheduling host selection response times (in milliseconds) as a function of reply set size. Host loads vary from 0 to 10%.

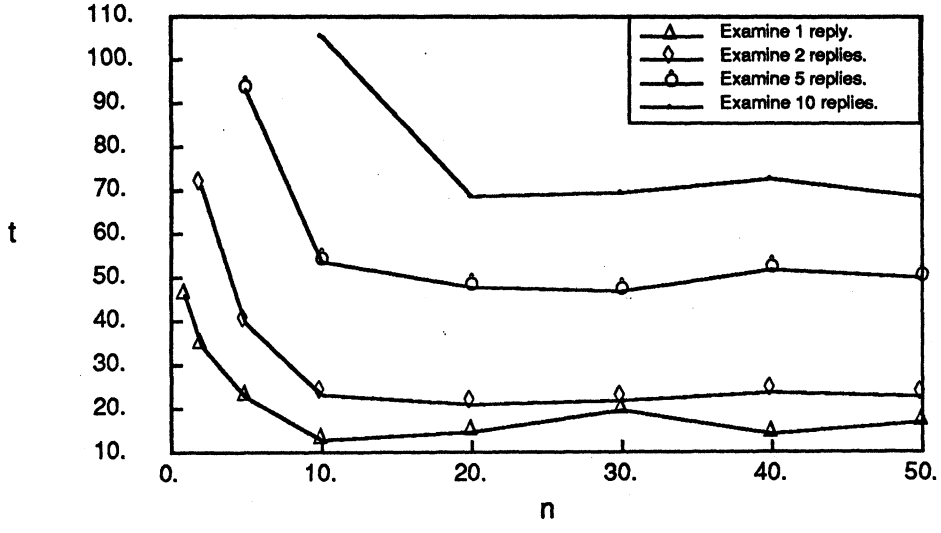

Figure 7.6: Decentralized scheduling host selection response times (in milliseconds) as a function of reply set size. Host loads vary from 0 to 100%.

Repliers	Replies lost (%)
1	0
2	3.8
3	30.0
4	45.0

Table 7.4: Multicast packet loss rates.

tunately, the granularity of the V-System's timing facilities is only 10 msecs; forcing the resultant reply delay interval to range between 10 msecs and 100 msecs. Of the replies that are returned to a query, the first r are used by the sender to make a scheduling selection and the rest are discarded without examination. Figures 7.5 and 7.6 show how the host selection time depends on the number of members in the light-load process group and on the number of replies actually used to make a host selection. Figure 7.5 corresponds to a system in which there are enough "idle" machines so that the reply set consists only of relatively unloaded machines. This is achieved by running response time tests in which all (simulated) program managers return loads between 0 and 10% processor utilization. Figure 7.6 corresponds to a system that is fairly heavily loaded, so that the reply set contains a significant number of machines that are not "idle". This is achieved by running the response time tests with all program managers returning loads that are uniformly distributed between 0 and 100%. Note that since our actual testbed system is typically very lightly loaded, the response times obtained in practice correspond to those of figure 7.5.

Several interesting things are represented in figures 7.5 and 7.6:

- The response time for examining a particular number of replies rises only gradually as the size of the reply set increases over the range measured. This is because once the enquirer no longer wishes to examine additional replies, he can let the operating system kernel internally discard all additional reply messages that arrive.
- The heavy penalty incurred for each additional reply message examined (about 5 to 10 msecs) is due to the fact that the kernel must receive and buffer all incoming reply messages until the enquirer terminates the multicast message transaction. Since many replies are returned, all increases in message transaction time will be significantly magnified.

  Fortunately, one must only look at a few replies in order to achieve good host selection results, as justified in Chapter 5. This claim has also been confirmed by running both the centralized and the decentralized scheduling facilities in parallel and comparing their selections. For example, Table 7.5 in Section 7.4.3 illustrates the results of running the same workload experiment using both our centralized and our decentralized scheduling facilities.
- The long response times at low reply set sizes are mainly due to the fact that significant packet loss can occur even at relatively low reply set sizes. Table 7.4, which is abstracted from the paper by Cheriton and Zwaenepoel [CZ85], illustrates this fact. Also, in figure 7.6, the use of a load distribution that is uniformly distributed over the entire range from 0 to 100% implies that the probability of having only high loads when only a few machines are involved becomes significant.

7.4.3 Load Metrics

Since the implementation of remote execution effectively separates the interactive and batch components of the system's workload by forcing an explicit decision about whether to remotely execute a program or not, both the centralized and the decentralized scheduling

Scheduling algorithm	Av. time to finish test suite (secs).	Standard deviation (secs)
Centralized, using processor utilization.	1230	55
Decentralized, using processor utilization.	1205	53
Centralized, using guest program count.	1274	157
Decentralized, using guest program count.	1202	73

Table 7.5: Comparison of processor utilization and guest program count as load metrics for a least loaded host scheduling algorithm.

facilities were configured to use either processor utilization or guest program count as their load metric for selection. In order to see how these metrics compared against each other an experiment was performed in which an identical workload was run using each of the load metrics with each of the scheduling facilities. The evaluation measure used for comparison was the throughput of the system, namely the amount of real time it took to finish the programs that comprised the workload.

In order to avoid environmental variations as much as possible, the experiments were run on a fixed set of workstations not being used by anyone else, and late at night—so that the load on the network file servers used was mostly due to the programs being run as part of the experiments. The workload itself consisted of programs that mimicked the actions of users by executing command scripts of typical sequences of remotely executed programs. In order to obtain a load sufficiently high that there was competition for resources the command scripts excluded the interactive activities, such as text editing, that would normally be interspersed between successive invocations of remotely executing programs, such as compilers. The specific set of activities included in the workload composition included:

- Compilation and timestamp consistency checking among the modules of a program. (Specifically, compilation of C programs and an extended version of the UNIX `make` program.)
- Searching for strings in files and comparing different files against each other. (The UNIX `grep` and `diff` programs.)
- Querying the operating system for various forms of local host state information and querying the distributed system for global state information (such as the set of all programs running in the entire system).
- Text formatting. (Specifically, the `TeX` text formatting program.)

Table 7.5 shows the results of these experiments. As can be seen from the table, guest program count performs essentially the same as processor utilization as a load metric.

7.4.4 Evaluation

The overall evaluation of centralized and decentralized scheduling has already been presented in Chapters 5 and 6 since the general design considerations of global scheduling are heavily influenced by the quantitative aspects of the hardware and software available in a distributed computer system. As has been repeatedly mentioned, the fledgling nature of our system with respect to size and workload has forced much of our evaluation of global scheduling issues to be based on extrapolation. This has been especially the case with

respect to fairness and load balancing considerations because we currently do not run a sufficiently heavy workload to make either an issue. However, one can certainly conclude that the facilities provided have proved more than adequate for the current system.

The central scheduler was implemented first because it did not require multicast communication facilities. However, the lack of migration facilities and a lack of reliable, dedicated server machines caused the central scheduler to be frequently unavailable. This reliability problem was the incentive that prompted investigation of decentralized scheduling facilities soon after multicast IPC became available.

7.5 Summary and Conclusions

In this chapter we have described an implementation of preemptable remote execution facilities that has allowed a quantitative assessment of "basic" remote execution facilities, migration facilities excluding machine-relative file and name servers, and global scheduling facilities. The implementation exhibited no noticeable problems of interference of migration with normal system operation and showed that typical programs (less than 1 Mbyte in size) can be migrated in a matter of a few seconds on hardware consisting of 1 Mips processors connected by a multi-megabit network. Our testbed also demonstrates that the complexity problems of migrating state among machine-relative file servers can be satisfactorily avoided by relying only on network file servers. Finally, our scheduling implementations have provided the quantitative input upon which the results of Chapters 5 and 6 are based.

In the diskless workstation-oriented V-System the cost of both local and remote program loading is dominated by the time to load a program's executable binary image from a network file server and, as a result, the two are essentially indistinguishable to the user. The V-System's emphasis on a system-wide I/O protocol and its support for network graphics has resulted in network-transparent program I/O, making the execution of local and remote programs also transparent to the user in most cases. Although the time to migrate a program is dominated by the time to copy its entire address spaces between two machines, which may take on the order of seconds, the interval during which the migrating program must be frozen can be reduced through pre-copying to the time needed to copy the program's "dirty-page" working set, which is a few tens to a few hundred milliseconds. Since remote execution only selects hosts that have had no recent local activity (in our case no local user at all), the few seconds it takes for a program to actually leave a machine is not a problem. Since users concurrently obtain a share of the CPU during migration, the actual operation is typically completely transparent.

The V-System and the hardware available in our system have proven to be a good basis for the work undertaken. The hardware used and the V-System's emphasis on fast remote IPC has allowed us to largely ignore the question of location of objects because the difference between local and remote IPC is only about a factor of three. Thus, network file servers have been used with impunity and issues of caching files on local disks have so far been ignored.

The availability of efficient *user-level* multicast facilities also proved to be an important benefit in that it allowed implementation of all of the global scheduling facilities outside the kernel of the operating system. Different approaches and implementations could easily be tested and interchanged. Similarly, the provision of well-known process groups provided a simple means by which machine-relative services could be referenced in a location-independent manner without having to add an additional form of machine-relative IPC referencing.

An important aspect of our facilities has been encapsulation of programs within separate address spaces with all communications with the rest of the world taking place through controlled IPC facilities. This has allowed users to let guest programs execute on

their machines without fear that they might corrupt the state of the operating system or machine.

An important simplification made in our system has been to use only network file servers. This has allowed us to avoid the most difficult issues described in Chapter 4 for dealing with residual dependencies. There are no machine-relative servers in the V-System that have large internal states requiring specialized internal migration support. There are also no machine-relative servers that can experience name conflicts. Perhaps most importantly, this has meant that we didn't have to rewrite our file servers—which would have been a major undertaking. We were able to take this approach because remote file access, even for most temporary file storage, proved quite satisfactory[LZCZ84,Zwa84]. In general, we espouse the principle that one should, to the degree possible, place the state of a program's execution environment either in its address spaces or in global servers in order to avoid the issue of residual dependencies altogether.

The time required to find a host candidate for remote execution or migration is minute compared to the time needed to perform the rest of either operation, being on the order of 13 milliseconds for both our centralized and decentralized implementations. Although we can only extrapolate scalability of our implementation, the performance numbers observed indicate that both our centralized and decentralized implementations should scale well to at least several hundred machines. Similarly, fairness and load balancing have not been an issue in our relatively small system, enabling us to only speculate on the effect of increased workloads and larger system sizes. However, since programs must be explicitly executed remotely and our experiments have shown that guest program count is a good metric if only batch programs are executed remotely, our design is in a good position to support the load balancing scheme described in Chapter 6 for providing distributed applications that are sensitive to unequal load distribution with approximately equal shares of the system's CPU resources.

The time to preempt a 1 Mbyte program in our system is a little over 4 seconds. If one assumes that users are willing to wait at least 10 seconds for preemption to finish, especially if they receive shared service during that time, then we can afford to do migration on machines of half the processing power. This would include most current workstations and personal computers, such as the IBM-PC/AT. Thus, we conclude that processor power is not a problem.

Another interesting hardware-related aspect of migration is the prevalence of large memories. These have allowed network paging to be treated as reasonable because of the very small paging frequencies needed with large local memories. Thus, the idea of avoiding local file/storage servers by relying strictly on network file/storage servers becomes more and more reasonable as memory cost reductions outpace those of all other forms of computer hardware[OCH*85].

Chapter 8

Conclusions and Future Work

In this thesis we have examined the issues surrounding the design and performance of preemptable remote execution facilities for a distributed computer system. We have shown that transparent preemptable remote program execution is feasible to provide, allowing idle machines to be used as a "pool of processors" without interfering with use by their owners and without significant overhead for the normal execution of programs.

Such facilities reduce the need for dedicated "computation servers" and higher-performance machines by providing one of the main benefits derived from traditional uniprocessor time-sharing systems, namely resource sharing at the processor and memory level. Moreover, they do so at a modest cost. Yet they do not preclude having specialized machines for floating point-intensive programs and other specialized computations; such dedicated server machines would simply appear as additional free processors. However, along with the functionality of global time-sharing capabilities come many of the design requirements of multi-user time-sharing systems, such as protection and global scheduling. We have shown how these can be provided in the context of a distributed operating system.

Four major issues have been addressed in our study of preemptable remote execution facilities:

- Programs must see a network-transparent execution environment.
- The interference with the system's normal operation that is caused by migration must be minimized.
- Residual dependencies in the form of state information in machine-relative servers must be migrated along with a migrating program.
- Global scheduling facilities must be provided to allow selection of appropriate remote hosts for program execution and migration.

In addition, we have shown how to provide specialized global scheduling policies and load balancing facilities to support distributed applications whose runtimes are overly sensitive to variations in the resource shares received by each of their subprograms. Our conclusions concerning each of these topics are presented in the following sections, which are arranged by function of the facilities provided.

8.1 Remote Execution

Given sufficient protection mechanisms to prevent programs from interfering with each other and the operating system, "basic" remote execution can provide many of the benefits put forth at the beginning of this thesis. If everyone were to execute their non-interactive tasks on the "least loaded host", then preemption through migration is needed mainly to deal with long-running servers and to prevent situations in which a local user's activities

would lock out the remotely executing "guest" programs running on the same machine. These problems are minimized by a programming paradigm that favors many short-term programs over longer-running ones that perform several tasks.

The principal requirements for supporting transparent remote execution, of the form defined in Chapter 1, are threefold:

- Programs must see an execution environment that is mostly network-transparent (see Section 1.2.3 for a definition of what we mean by "mostly"). This typically implies interaction with the rest of the system through network-transparent IPC and a global naming space. The concept of location must be preserved in order to refer to such things as the I/O instances associated with a particular user or the temporary files created by the invoker of a program. But locality must be based on references to logical entities rather than physical ones.
- The distributed operating system must provide many of the features of a distributed multi-user time-sharing system. Specifically, it must provide adequate protection and authentication among users' programs. (As discussed elsewhere, it must also provide reasonable and fair access to the resources of the distributed system by means of suitable global scheduling facilities.)
- Programs must be able to access all physical objects, such as I/O devices, in a network-transparent manner. This typically means using network-transparent IPC to communicate with "front-end" servers that are co-resident with the hardware they manage.[1]

Our experiences with providing remote execution for the V-System lead us to believe that its provision is not difficult in the context of any distributed operating system that employs network-transparent IPC and encapsulation of programs in separate address spaces so that they can only communicate via the network-transparent IPC.

However, our implementation has also benefited from the V-System's extremely efficient implementation of network communications, which typically makes remote execution network-transparent with respect to performance, in addition to being semantically transparent. If the cost ratio of remote-to-local communication were a factor of 10 or 20 instead of 3, and workstations all had local disks, then one might wish to consider more complicated scheduling designs that take into account the locality of files needed for program loading and/or execution.[2] This leads us to conclude that a "well-balanced" environment in which local and remote communication are similar in cost is a very desirable goal. This requires efficient low-overhead network communication protocols as well as augmentation of high-performance machines with equally high-performance network interfaces that avoid making network communications a bottleneck.

8.2 Preemption and Migration

If running guest programs concurrent with local users' activities on a machine is unacceptable, then transparent migration is needed to support preemption without undue loss of work. While constraints such as hard limits on memory resources or failure considerations such as impending reboot of a machine's operating system provide examples of situations where migration is clearly necessary, our experience with real users has been that a surprisingly important consideration to them is simply the ability to exert absolute authority over their private domain of facilities. This attitude appears most prevalent with respect to personal workstations, which many users seem to view with much the same attitude as

[1]Such servers may be provided either as separate user-level processes or as part of the local operating system kernel.

[2]Most existing distributed operating systems fall closer to the factor of 3 than to the factor of 10 or 20.

they do their private car: they may be willing to loan their car to a friend but are rarely willing to carpool together and demand the authority to claim exclusive usage at any time.

The other main benefit we perceive from support of migration is the ability to provide various forms of load balancing and load sharing:

- Preemptive migration represents a form of load balancing that prevents resource contention between guest programs and significant local activities. This is especially useful for longer-running programs that will thereby maintain access to a reasonable share of resources rather than being "starved" by privileged local activities.
- Migration allows us to run global servers, such as authentication servers and configuration monitoring servers, on general-purpose machines rather than dedicated server machines. While most user programs are generally short-term in duration and could usually tolerate the diminished resource shares that would result if they had to share a machine with privileged local activities, global servers are intended to run forever and are typically sensitive to reductions in their resource allocations. Note that network file servers can easily become system performance bottlenecks and, hence, shouldn't be used to run miscellaneous other servers. While the cost of additional server machines may not be significant for larger systems, it can be important to smaller installations in which the cost of an extra one or two machines is a non-trivial fraction of the system's capital costs.
- Migration allows us to implement a load balancing scheme that, together with an appropriate global scheduling policy for host selection, provides approximately equal CPU shares to the subprograms of distributed applications whose subprograms are independent of each other. This is important for applications that are sensitive to variations in the resource allocations their subprograms receive, such as applications that run at the speed of their slowest subprograms.

Note that while these examples represent specific instances in which load balancing/sharing are useful, we have not attempted to address issues of the general utility of load balancing facilities in this thesis.

Two major issues have been addressed in the provision of migration facilities:

- Minimization of the interference caused by migration to the normal operation of the system as a whole and to migrating programs in particular.
- Proper handling of residual dependencies to machine-relative servers that occur when a program migrates.

Interference can be minimized by using a technique we call *pre-copying*. With pre-copying, program execution and operations on the program by other processes are suspended only for the last portion of the copying of the program's state rather than for the entire copy time. In particular, critical system servers, such as file servers, are not subjected to inordinate delays when communicating with a migrating program. In most cases pre-copying can reduce the suspension time of a migrating program to a time interval of similar length to that incurred if the program were to be swapped out to secondary storage in a paged virtual memory system. Since the system must already handle communication delays of this sort, the interference caused by migration should usually be negligible. In contrast, if pre-copying were not used, then the resulting delays imposed on communications and higher-level operations may be several orders of magnitude larger than normal. For example, in the V-System it takes about 3 seconds/Mbyte to copy the address spaces of a program between two SUN-2 workstations connected by a 10 Mbit Ethernet, whereas message retransmission timeouts are measured in fractions of a second.

Another aspect of handling interference depends on the operating system kernel providing reliable communication facilities that can queue message packets in order to relax the real-time constraints of communication during the time interval that a migrating pro-

gram is frozen. An important aspect of this is that the kernel must queue both reliable *and unreliable* messages in order to preserve statistical communication characteristics that programs may depend on at a higher level. While the short suspension times achieved through pre-copying make this point relatively unimportant for unicast communications, it can severely affect multicast communications in which many unreliable reply messages may arrive for a sender within a very short period of time.

Removing residual dependencies caused by migration of a program requires migrating state information stored for it in machine-relative servers on the original host the program was running on. This may be difficult because some machine-relative servers contain *large* amounts of state information. The most notable of such servers are file servers. In order to minimize the interference caused by migration such servers must implement the same pre-copy techniques employed to copy the state of a program's address spaces, implying that they must incorporate migration support into their basic structure rather than simply providing generalized query/install operations. While this is not excessively difficult to implement, it does imply significant alteration of servers that are typically complicated to begin with.

Another implication for machine-relative servers that implement named objects is that their name spaces must take on a global scope in order to accommodate objects that exist beyond the demise of their creators and, hence, cannot be kept in program-relative contexts to prevent naming conflicts caused by migration.

We advocate that one should avoid the residual dependencies arising from migration, to the greatest degree possible, by placing the state of a program's execution environment either in its address spaces or in global servers. That way the state is either migrated to the new host automatically as part of copying the address spaces or it need not be moved at all. Experience with name bindings in V, which are stored in a cache in each program's address spaces as well as in global servers, provides another example of how this can be successfully done. Of course, the architecture of a system will determine how easy it is to follow this advice. For example, if every machine has a local disk, then it may be difficult to avoid a demand for machine-relative file servers.

As mentioned, however, not all machine-relative servers can be easily eliminated. This implies two additional requirements:

- The IPC facilities must support the concept of *well-known* identifiers, so that references to machine-relative servers can be rebound to *different* machine-relative servers on another machine. In the V-System this was achieved with the more general-purpose facility of well-known multicast process groups. In systems that support kernel-protected communication ports or links, such as Accent or Demos-MP, the use of well-known port or link indices can achieve the same effect.
- Machine-relative servers must structure those operations that might modify the state of a frozen program so that they can be deferred and correctly forwarded when the program migrates. Depending on whether the reference rebinding mechanism employed can suppress duplicates, care must be taken to deal correctly with nonidempotent operations during their forwarding.

Finally, migration of a program requires transparent access to and control of its entire state, implying a protection and authorization scheme in the operating system that allows this. Whereas, for example, debugging also requires the same level of access to a program's state, if only debugging needed to be supported, then a protection scheme based on the concept of a single privileged owner process can suffice. Supporting migration implies that two separate processing entities may need simultaneous access to a program's state, and, therefore, that an authorization scheme is needed that allows access privileges to be conferred to multiple processing agents.

8.3 Global Scheduling

Design considerations and experience with the V-System have indicated that both centralized and decentralized designs for global scheduling can be made to work well and should be scalable to systems containing hundreds of machines. Although we can only conjecture what future workloads will be like, we claim that the designs described in this thesis should be fairly robust with respect to scaling. We argue that other considerations, such as network bandwidth, are more likely to dominate the scalability of a system than global scheduling constraints.

The principal tradeoff we perceive between centralized and decentralized designs is one of flexibility versus simplicity. A centralized design can be achieved without the availability of multicast, whereas a decentralized design cannot. A centralized design can be made to accommodate larger sized systems than a decentralized design can, and if multicast is used, can achieve essentially the same level of reliability/availability as a decentralized design. The advantage of the decentralized design we have proposed is its simplicity: Multicast is used to query a set of likely machine candidates and statistical sampling arguments are invoked to justify only looking at a small sample of the returning replies, thereby avoiding the flooding problems that normally result when many replies are returned to a single query.

The approach we advocate to issues of fairness and load balancing is to do nothing. If one assumes that users generate comparable workloads in the long run, then statistically all users will receive equal access to the system's resources. Local multi-tasking, combined with preemptive migration, ensures that guest programs will receive at least some share of the system's CPU resources at all times. We claim that this level of fairness and load balancing is sufficient in most cases and that anything more in the way of general facilities would be difficult and expensive to obtain. UNIX, which also does not enforce fairness constraints between programs on a single machine, has demonstrated that social pressure seems effective enough to deal with users who might feel inclined to abuse their privilege to start arbitrary numbers of programs running in parallel.

The principal case where unequal resource shares among programs becomes an issue is for distributed applications that depend on equalized resource shares among their subprograms for efficient execution. (Applications that depend on *guaranteed* resource allocations for their correct execution, such as benchmark programs, are another case where a "do-nothing" policy is insufficient. This topic is discussed further in Section 8.5.2.) The differences in CPU shares received by each remotely executing program in the system can be limited by using a global scheduling policy based on a least loaded host selection algorithm that uses guest program count as its load metric. The scheduling policy, which assumes preemptive migration of guest programs and round-robin scheduling of guest programs on each machine, provides roughly equal CPU shares to all remotely executing programs throughout the system. As long as the number of remotely executing programs in the system does not exceed the average number of such programs by more than a factor of two, every machine will have at most twice as many guest programs running on it than the average number throughout the system. This guarantees that subprograms will receive at worst half the CPU share that is provided to the average guest program in the system. If the cost of migration is small relative to the runtimes of typical programs, then migration can be used to deal with temporary load variations that exceed a factor of two by migrating programs from excessively heavily loaded machines, implying that the bound of a factor of two can almost always be achieved.

For the important class of distributed applications whose subprograms are independent of each other and, hence, do not need to communicate with each other, an explicit load balancing scheme can be used to ensure even tighter bounds on CPU shares. Starting from the lower bound on CPU share provided each subprogram by the global scheduling policy just mentioned, simply migrating subprograms off heavily loaded machines onto lightly

loaded machines as they become available (e.g. as the subprograms running on them finish) will ensure that the termination times of all subprograms will be within 17% of that resulting from somehow giving all subprograms equal CPU shares. Actual completion times will be slower by the time required to migrate subprograms. If we assume that *preemptive* migration occurs infrequently enough to be ignored in our timing considerations, then each subprogram will be migrated at most once due to our load balancing scheme. Whereas improvements on this load balancing scheme might be achieved by making assumptions about the runtimes of programs, it is unclear whether the improvements would be worth the effort given that they would very likely require application-specific assumptions.

An important result relevant to this load balancing scheme is that the load metric of guest program count is a reasonable one to use if the interactive and batch components of the system's workload can be separated. In fact, in this case it is a slightly better selection metric to use than processor utilization—which must be averaged over some period of time and, hence, suffers from latency problems.

8.4 General Considerations

If there has been a general theme throughout this thesis, it has been to avoid adding facilities wherever possible and thereby to keep things simple. Remote execution forces systems to face the complexities of truly distributed applications head-on and migration is complex by nature. In this thesis we have shown how a system must be structured to support these facilities and we have demonstrated how to solve the attendant problems of performance and interference.

However, difficult problems still remain that center around managing the complexity of extracting the state of a program from its environment. As a result our conclusions must be presented with a caveat: While preemptable remote execution is perfectly feasible to provide, it may extract a significant price in terms of added system complexity. The magnitude of this price depends on how much of the system can be isolated from having to know about remote execution or migration. In our implementation for the V-System we were able to obtain significant simplifications from the fact that naming and file service did not require machine-relative servers and, hence, became independent of migration. We have advocated a decentralized scheduling design because of its greater simplicity and we have argued against any explicit facilities for dealing with fairness and load balancing. But we succeeded only marginally in providing kernel-level support for migration in a non-fragile manner that did not produce maintainability problems.

By not choosing the most general design possible, a considerably simpler, and, hence, more tractable approach to the problems involved has been achieved. It seems likely that the success of other efforts to provide remote execution of programs, with or without preemptive migration, will depend significantly on how well they can achieve similar simplifications in order to provide the capabilities they really need and no more.

8.5 Future Work

A number of questions remain to be resolved about preemptable remote execution. In this section we will examine several possible avenues of future work.

8.5.1 Usage Considerations

Several questions have been raised throughout this thesis but left partially unanswered with respect to how (preemptable) remote execution facilities should actually be used. These can be collapsed, for the most part, into a single question about the nature of the workload we

expect to run on distributed computer systems in the future. Our own experience with the V-System has been with a workload consisting primarily of non-distributed applications that fall into one of two categories: long-running interactive programs such as various text and picture editors, and "fast-turnaround" batch jobs such as compilation and text formatting. This workload has fit very naturally into a usage paradigm wherein interactive programs are executed locally and batch jobs are executed in parallel on remote machines. Since our interactive applications do not perform much "background" computation, the problem of overloading the local machine with several concurrent applications has not been a problem. One could imagine that more sophisticated applications might violate this assumption.

An alternative approach to program execution is to execute *all* programs on the "least loaded host", be it local or remote. Such a policy would address the question of overloading the local machine automatically and would also simplify the user interface in that users would no longer manually have to decide whether a program should be executed locally or remotely. But it would also now cause interactive programs to be executed remotely, bringing up the issues mentioned in Chapter 3 of how well highly interactive applications will perform in a remote setting. While the work done by Nowicki and Lantz has given indication that many application classes can be handled quite well in such a setting, more work is still needed to determine just how far such an approach can be taken.

The other area affected by workload assumptions is, of course, the work we have done on global scheduling issues and designs. Only actual experience with future workloads will tell whether the assumptions we have made will bear any relation to reality. Perhaps equally importantly, the *size* of the workload generated by each user may be the determining factor in whether or not preemptive migration is needed in order to make remote execution acceptable to the owners of the machines of a system. To date, our experience with the V-System has been that most machines are typically 80% idle and that there are frequently a few machines with no local users logged on. This has resulted in a situation where preemption is rarely needed, especially since guest programs are run at a lower scheduling priority than local ones. As mentioned, some users seem unwilling to coexist with guest programs, and if the workload were to increase significantly, migration would become necessary to prevent unacceptable contention for one machine's resources while other machines stand idle. But if the number of resources per user continues to stay at a high level, so that remote execution is used mainly to spread bursts of activity across multiple machines, then preemption facilities may not be worth the effort. An open question is whether that would be acceptable to users. An additional variable is the question of whether applications turn out to consist mainly of longer-running programs or of (several) relatively short programs, so that guest programs would disappear of their own relatively quickly.

8.5.2 Multiple Scheduling Domains

Chapter 6 presented a global scheduling policy and load balancing scheme that provides distributed applications whose subprograms don't communicate with each other with approximately equal CPU shares. While this class of applications is a large and important one, it is not the only one that is sensitive to variations in resource shares. Some applications require *guaranteed* resource shares to function correctly, or at least efficiently. Notable examples include benchmark programs and distributed applications that perform internal scheduling decisions based on assumptions about fixed resource distributions.

For these kinds of applications some sort of *resource reservation* scheme will be necessary to ensure desired behavior. This, in turn, raises questions of how such a scheme will interact with the autonomy assumptions we have made and the migration facilities that support them. While migration would probably invalidate the results of a benchmark program, it may not be a problem for a distributed application whose internal scheduling decisions can tolerate a *few* transient variations in load.

Provision of a resource reservation scheme would effectively allow the system to provide "multiple scheduling domains" since several schedulers could now be provided with their own sets of resources. An important question about such a system of multiple scheduling domains is how "separated" the domains should be for various classes of applications. By this we mean whether or not resources can reside in more than one scheduling domain and how dynamic the allocation of resources can be—under what conditions resources can be transferred/preempted between domains. Clearly, from the point of view of machine autonomy and system throughput efficiency considerations, it would be desirable to have as much overlap of scheduling domains as possible.

Separate from the question of what kinds of resource guarantees various applications need is the question of how to implement a resource reservation scheme in the first place. This involves several issues, which we briefly introduce here:

- Applications will somehow have to specify their resource reservations and the conditions under which they are willing to be preempted. For example, an application might simply specify that it desires a guarantee of always receiving at least some specified share of resources, with no indication given of whether migration is acceptable or not. A more appropriate specification for a preemptable environment would specify under what circumstances migration of subprograms would be acceptable and appropriate.
- If resources are reserved on a machine before an application starts a subprogram running there, then one might wish to allow the reservation to be preempted to another machine. If so, we must provide a means by which applications can *find* their reserved resources independent of their location.
- In general, there is a question of how schedulers find and communicate with the resources they control. Multicast will likely play an important role in allowing schedulers to communicate with the entities in their domain and no others.
- One might wish to provide some sort of authentication scheme to ensure that reserved resources are only accessible to programs from the scheduling domain that reserved them. An added wrinkle is that such programs may come into existence at a later point in time than when the resources were reserved.

Finally, one must determine what level of support for such scheduling domains is justified on the basis of what kinds of workload a given system actually intends to run.

8.5.3 Extension to an Internet Environment

In this thesis we have limited ourselves to considering only a single local network as the domain over which programs can be remotely executed or migrated. While this restriction was imposed mainly in order to be able to assume efficient multicast support, it has also allowed us to ignore questions of communication performance in an internet setting. If the speed of file loading or copying of address spaces is significantly slowed because of intervening gateway machines and/or slow communications links, then many of our performance assumptions are complicated. While Nowicki and Lantz's work has indicated that one can still achieve acceptable performance for network I/O in such an environment through appropriate structuring of the protocols and front-end servers used, we must still face scheduling questions of how to take into account file transfer costs when considering remotely executing a program on a particular host candidate. For migration we must similarly worry about the cost tradeoffs of migrating within a local network versus migrating a program to another network. It may not even be feasible to copy the program's address spaces quickly enough to satisfy the preempting party's time constraints if the network links used are too slow.

Perhaps a more important question to address is whether remote execution and mi-

gration across a general internet is really the desired goal in the first place. One can reasonably argue that remote execution to a Cray machine on a distant network would be a very nice capability to support, although one might also argue that the likelihood of such a remote system allowing transparent execution access to itself will be small. The case for migration is even more strained since most machines that are willing to accept service requests from remote sources are probably dedicated server machines that won't preempt the tasks they have accepted to do.

A much more likely environment to consider is a *local internet*, which we define as an internet consisting of a few (high-speed) local area networks connected by reasonably powerful gateway machines. Such a local internet represents the network environment seen within a domain such as a university campus, a business installation, or other similar-sized entities. It represents a domain that is frequently controlled by a single administrative authority, or several cooperating ones—a domain over which machines might reasonably be expected to cooperate with each other on issues such as authentication and resource sharing. Most importantly for us, it represents a domain that is small enough to allow us to make assumptions such as reasonably high-speed communications between all machines within the system and perhaps even to assume that multicast can be efficiently implemented within it (for example, by having gateways forward multicast packets to adjacent networks).

If multicast is not supported in such an environment, one must employ the modified designs we have described for rebinding references to migrated entities and must use a global scheduling scheme that either employs hierarchical schedulers or makes use of a central scheduler that doesn't rely on multicast. For the V-System we would also have to modify the approach used for name resolution. While these restrictions may be annoying, one can envision ways of building a system that could accommodate them. For example, one way is to simply have a single global scheduler and a single reference rebinding "registrar" for the entire local internet. This would be the simplest design to implement, but would not provide any optimization for what will most likely be the common case of remote execution and migration within each individual network. The alternative is to provide an instance of these two servers on each local network, which then cooperate with their counterparts on other networks to exchange information as needed. Aside from removing delays through gateways and multiple networks from the access path to a critical system service, such a design could also be added transparently to an existing design for a single local network since the servers could respond to appropriate multicast rebinding and scheduling messages as needed. If the system's naming facilities are also based on multicast, then a similar "transparent server" approach will be needed for naming. Open questions that must be resolved include an evaluation of exactly how well such servers will perform for various system sizes and local/remote distributions of programs throughout the local internet and how often they should exchange information among themselves.

An alternative approach to dealing with a local internet is to extend the system's multicast facilities to cover the entire domain. Deering and Cheriton[DC85,CD85], are investigating ways of supporting multicast within a general internet environment and several people have suggested an approach to multicast that is based on the concept of concentric "rings of effectiveness". The latter approach is one in which multicast messages include a minimum and maximum range of effectiveness based on, for example, the number of network hops a message has gone from its source. Thus, messages could be sent to the local network, all networks one gateway away from the sender's network, all networks two gateways away from the sender's network, and so on. If the internet is multiply connected so that cycles can occur, then one might wish to implement some sort of control in the gateway hosts to avoid multiple instances of a packet being generated for a given network. On the other hand, if multicast queries rarely have to be sent beyond the immediate network, then it may be more appropriate to simply have all receivers implement their own duplicate suppression (for example, using sequence numbers).

Using an approach in which senders iteratively send out their message to an increasingly larger ring of receivers would allow us to transparently extend a local network version of our facilities while still bounding the multicast traffic of the system by the size of each local network instead of the size of the entire local internet. However, this approach also implies that the time to *fail* in finding a desired response to a multicast query is substantially increased since senders must wait for multiple query messages to fail in delivering a response. For reference rebinding to migrated objects and global scheduling queries one can argue that failed queries should not occur very often. It is less clear how facilities such as name resolution would be affected by this issue.

8.5.4 Failure Recovery Issues

If failure recovery facilities are available, then an alternative approach to preemption of a program is to simply destroy it and rely on the system's failure recovery facilities to recover it onto another machine. This would be the most appropriate way of dealing with preemption if failure recovery support is needed for programs anyway. However, failure recovery is typically a very "heavy-weight" capability to support, involving at least as much overhead and complexity of design as would be needed to support just program migration. Since our principal goal with preemptable remote execution is to gain access to additional resources, we must be careful not to waste more resources in the process than are gained. For example, if we spend more time checkpointing a program than would be needed to simply kill it and run it again (assuming that it is idempotent), then we have achieved no benefit from remote execution whatsoever.

The real question is thus one of workload nature for a system. The remote workload we run in our own testbed V-System consists primarily of idempotent applications such as compilations and text formatting. Failure recovery for those applications, such as editors, that manipulate non-reconstructable state is provided by each application individually, mainly in the form of periodically invoked checkpoints of relevant state. For such a system, general purpose failure recovery facilities have not seemed appropriate, especially since the hardware base for our system rarely has failures of machines to begin with. In a system in which most applications require failure recovery support, for example in a database-oriented system, general failure recovery facilities might be much more appropriate, making the tradeoff between provision of full failure recovery support and just program migration go the other way.

One can also ask the converse question: Are program migration facilities useful for implementing failure recovery support? If a "checkpoint/restart server" mimics the appropriate actions, then checkpoints and restarts of programs from checkpoints can be treated as special cases of migrating a program's state in which one end of the migration path is the checkpoint/restart server. Thus, one approach to provision of general-purpose failure recovery facilities would be to periodically "migrate" the state of a program to a checkpoint/restart server and to use the stored state to "migrate" the program to another machine should the first one fail.

Unfortunately, there are several rather severe problems with such an approach. The principal problem is that application-transparent failure recovery schemes can only restart a program at its point of failure since they have no idea of what important actions have occurred since the last checkpoint they performed. This implies that some sort of "roll forward" ability must also be implemented in addition to simply providing a checkpoint/restart facility. Various people have devised schemes for transparently recording the external interactions of programs so that a program can be restarted from its last checkpoint and then "executed forward" to its point of failure by feeding it the correct inputs and discarding its (duplicated) output appropriately[PP83,BBG83]. However these schemes rely on special hardware support or require that every external interaction of a program (every I/O operation and every message sent and received) be performed as a

transaction so that it is reliably recorded.

Thus, provision of a "roll forward" capability in support of transparent failure recovery is considerably more difficult than one might expect and may also be considerably more expensive in terms of performance degradation than is acceptable. The alternative is to provide application-visible failure recovery support so that programs can cause checkpoints of their state to be performed at "clean points", such that restart can be performed directly from a checkpoint without the need for a roll forward capability. While this could still be done using the facilities provided for program migration, one must now ask whether there is any point in checkpointing the entire program state instead of just its important state variables. There are two reasons why checkpointing the entire state of a program may not be appropriate:

- Checkpointing the entire state of a large program is expensive, possibly taking several seconds to copy its entire state.[3] Furthermore, unless pre-copying can somehow be done on a continual basis, we cannot employ pre-copying techniques to reduce this time since the checkpoint must be invoked by the program itself at its clean points and the program must then wait for the checkpoint to finish before continuing execution. Having a program manually checkpoint just its important state variables should take considerably less time than this.

  If programs are demand-paged onto a reliable backing store, then one possibility for reducing the time to checkpoint an entire program is to store the checkpointing information intertwined with the backing store copy of the program. Migration, and, hence, also checkpoints, would now involve flushing a program's dirty pages out onto the backing store and the time required to do so would be a function of the program's working set size and the operating system's page replacement policies. In order to maintain both correct checkpoint information as well as a current backing store copy of the program, the backing store would have to maintain duplicate pages for each page of the program's state that has been modified since the last checkpoint taken.

- Perhaps a bigger problem with our approach to checkpointing a program's state is that it may not be feasible, in general, to restart it transparently. This is due to the transient nature of some program state, such as open file instances. As was the case with interference considerations for program migration, we must worry about preventing the rest of the system from timing out various connections and transactions before the failed program can be restarted. Since we cannot rely on pre-copying techniques to reduce the restart time this issue remains an open problem, perhaps requiring a change in the way that applications deal with timing considerations. One can envision that applications that already have to invoke their own checkpoints will also be willing to check with some system failure recovery service before deciding to abort their connections with a failed party.

In conclusion, while it may be the case that program migration facilities will turn out to be useful tools for implementing failure recovery support, they will almost certainly represent only a small part of the solution rather than a significant fraction thereof. On the other hand, the possibility of amortizing the overhead of such facilities by employing them for both migration and failure recovery duties warrants further investigation.

[3] This ignores any time overhead needed to ensure that the checkpoint information is stored on stable storage instead of just in the main memory of the recording machine.

Bibliography

[ACF84] Y. Artsy, H.Y. Chang, and R. Finkel.
Charlotte: Design and Implementation of a Distributed Kernel.
Computer Sciences Tech. Report 554, Univ. of Wisconsin at Madison, August 1984.

[AE85] R. Agrawal and A.K. Ezzat.
Processor sharing in nest: a network of computer workstations.
In *1st International Conference on Computer Workstations*, pages 198–208, IEEE, November 1985.

[AHL*80] C.J. Antonelli, L.S. Hamilton, P.M. Lu, J.J. Wallace, and K. Yueh.
Sds/net - an interactive distributed operating system.
In *IEEE COMPCON Fall 80*, IEEE, September 1980.

[Bar81] J. Bartlett.
A nonstop kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 22–29, ACM, December 1981.

[BBG83] A. Borg, J. Baumbach, and S. Glazer.
A message system supporting fault tolerance.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 90–99, ACM, October 1983.
Published as *Operating Systems Review* 17(5).

[BF81] R.M. Bryant and R.A. Finkel.
A stable distributed scheduling algorithm.
In *Proceedings 2nd International Conference on Distributed Computer Systems*, 1981.

[BFL*76] J.E. Ball, J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner.
Rig, rochester's intelligent gateway: system overview.
IEEE Transactions on Software Engineering, SE-2(4):321–328, December 1976.
Reprinted in K.J. Thurber, editor, *Tutorial: Local Computer Networks* pages 316-323. IEEE Press, 1980. Also TR5, Department of Computer Science, University of Rochester.

[BHM77] F. Baskett, J.H. Howard, and J.T. Montague.
Task communication in DEMOS.

November 1977.
Published as *Operating Systems Review* 11(5).

[BL83] A. Barak and A. Litman.
Mos: a multicomputer distributed operating system.
1983.
In preparation.

[BP84] D.A. Butterfield and G.J. Popek.
Network tasking in the LOCUS distributed UNIX system.
In *Proc. Summer USENIX Conference*, pages 62–71, USENIX, June 1984.

[Cas81] L.M. Casey.
Decentralized scheduling.
Australian Computer Journal, 13:58–63, May 1981.

[CD83] J.H. Clark and T. Davis.
Workstation unites real-time graphics with UNIX, ethernet.
Electronics, 113–119, October 1983.

[CD85] D.R. Cheriton and S. Deering.
Host groups: a multicast extension for datagram internetworks.
In *Ninth Data Communications Symposium*, pages 172–184, IEEE, September 1985.

[Che84] D.R. Cheriton.
The V kernel: a software base for distributed systems.
IEEE Software, 1(2):19–42, April 1984.

[Che86] D.R. Cheriton.
A uniform i/o interface and protocol for distributed systems.
1986.
To appear in ACM Transactions on Computer Systems.

[CM85] D.R. Cheriton and T.P. Mann.
A decentralized naming facility.
1985.
Submitted for publication. Also available as Stanford Computer Science Dept. Tech. Report 1098.

[CMB*79] J.P. Cabanel, M.N. Marouane, R. Besbes, R.D. Sazbon, and A.K. Diarra.
A decentralized os model for ARAMIS distributed computer system.
In *Proceedings of the 1st International Conference on Distributed Computing Systems*, October 1979.

[Cra83] D.H. Craft.
Resource management in a decentralized system.
In *SOSP9*, pages 11–19, ACM, October 1983.
Proceedings published as *Operating System Review* 17(5).

[CS86] D.R. Cheriton and M. Stumm.
The multi-satellite star: structuring parallel computations for a workstation cluster.

1986.
Submitted for publication.

[CSB85] D.R. Cheriton, M. Stumm, and L. Berc.
Measurements of the V-System's multicast facilities.
1985.
Private communication.

[CZ85] D.R. Cheriton and W. Zwaenepoel.
Distributed process groups in the V kernel.
ACM Transactions on Computer Systems, 3(2):77-107, May 1985.
Presented at the SIGCOMM '84 Symposium on Communications Architectures and Protocols, ACM, June 1984.

[Dan82] R.B. Dannenberg.
Resource sharing in a network of personal computers.
PhD thesis, Carnegie-Mellon University, 1982.
Technical Report CMU-CS-82-152, Department of Computer Science.

[DC85] S. Deering and D.R. Cheriton.
Host groups: A Multicast Extension to the Internet protocol
RFC 966, Network Information Center, SRI International, December 1985.

[ELZ84] D.L. Eager, E.D. Lazowska, and J. Zahorjan.
Dynamic Load Sharing in Homogenous Distributed Systems.
Department of Computer Science Tech. Report 84-10-01, University of Washington, October 1984.

[Fin80] R. Finkel
The Arachne Kernel.
Computer Sciences Tech. Report 380, University of Wisconsin at Madison, April 1980.

[FL72] D.C. Farber and K.C. Larson.
The distributed computer system.
In *Proceedings Symposium on Computer Communication, Networks and Teletraffic,* 1972.

[FR85] R. Fitzgerald and R.F. Rashid.
The integration of virtual memory management and interprocess communications in accent.
In *Proc. 9th Symposium on Operating Systems Principles,* pages 13-15, ACM, December 1985.
Published in *Operating Systems Review* 19(5).

[GK85] R.D. Gaglianello and H.P. Katseff.
Meglos: an operating system for a multiprocessor environment.
In *Proc. 5th International Conference on Distributed Computing Systems,* pages 35-42, May 1985.

[Gro86] Distributed Systems Group.
V-System Reference Manual

Technical Report, Computer Systems Laboratory, Stanford University, 1986.

[Ham80] R.A. Hammond.
Experiences with the series/1 distributed system.
In *Proc. Fall COMPCON*, pages 585–589, IEEE, September 1980.

[How72] R.H. Howell.
The integrated computer network system.
In *Proceedings 1st International Conference on Computer Communications*, 1972.

[HW80] R.H. Halstead and S.A. Ward.
Munet: a scalable decentralized architecture for parallel computation.
In *Proceedings IEEE 7th Annual Symposium on Computer Architectures*, 1980.

[JP84] E.D. Jensen and N. Pleszkoch.
Archos: a physically dispersed operating system.
IEEE Distributed Processing Technical Committee Newsletter, 6(SI-2):15–24, June 1984.

[KG83] P. Kavaler and A. Greenspan.
Extending UNIX to local-area networks.
Mini-Micro Systems, September 1983.

[Kie81] R.G. Kieburtz.
A distributed operating system for the stony brook multicomputer.
In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, April 1981.

[Kle75] L. Kleinrock.
Queueing Systems Volume 1: Theory.
Wiley, 1975.

[Kle76] L. Kleinrock.
Queueing Systems Volume 2: Computer Applications.
Wiley, 1976.

[Lan81] G. Le Lann.
A distributed system for real-time transaction processing.
IEEE Computer, 14:43–48, February 1981.

[LGFR82] K.A. Lantz, K.D. Gradischnig, J.A. Feldman, and R.F. Rashid.
Rochester's intelligent gateway.
Computer, 15(10):54–68, October 1982.

[LM82] M. Livny and M. Melman.
Load balancing in homogeneous broadcast distributed systems.
In *Proceedings ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[LN84] K.A. Lantz and W.I. Nowicki.
Structured graphics for distributed systems.
ACM Transactions on Graphics, 3(1):23–51, January 1984.

[LSP82] L. Lamport, R. Shostak, and M. Pease.

The byzantine generals problem.
ACM Transactions on Programming Languages and Systems, 4(3):382–401, July 1982.

[LZCZ84] E.D. Lazowska, J. Zahorjan, D.R. Cheriton, and W. Zwaenepoel.
File access performance of diskless workstations.
Technical Report STAN-CS-84-1010, Department of Computer Science, Stanford University, June 1984.

[MB76] R.M. Metcalfe and D.R. Boggs.
Ethernet: distributed packet switching for local computer networks.
Communications of the ACM, 19(7):395–403, July 1976.

[MFH83] T.W. Malone, R.E. Fikes, and M.T. Howard.
Enterprise: A Market-like Task Scheduler for Distributed Computing Environments.
Technical Report, Xerox Palo Alto Research Center, October 1983.
Cognitive and Instructional Sciences Group.

[MP81] B. Miller and D. Presotto.
Xos: an operating system for the x-tree architecture.
Operating Systems Review, 15:21–32, 1981.

[NL80] D.A. Nowitz and M.E. Lesk.
Implementation of a dial-up network of UNIX systems.
In *IEEE COMPCON Fall 80*, pages 483–486, IEEE, September 1980.

[Now85] W.I. Nowicki.
Partitioning of Function in a Distributed Graphics System.
PhD thesis, Stanford University, 1985.

[OCH*85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson.
A trace-driven analysis of the UNIX 4.2 bsd file system.
In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–25, ACM, December 1985.

[PD80] R. Peebles and T. Dopirak.
Adapt: a query system.
In *Proceedings COMPCON*, Spring 1980.

[PM83] M.L. Powell and B.P. Miller.
Process migration in DEMOS/MP.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 110–119, ACM, October 1983.
Published as *Operating Systems Review* 17(5).

[Pow77] M.L. Powell.
The DEMOS file system.
In *Proc. 6th Symposium on Operating Systems Principles*, pages 33–42, ACM, November 1977.
Published as *Operating Systems Review* 11(5).

[PP83] M. Powell and D.L. Presotto.
Publishing: a reliable broadcast communication mechanism.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 100–109, ACM, October 1983.
Published as *Operating Systems Review* 17(5).

[PWC*81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel.
LOCUS: a network transparent, high reliability distributed system.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 169–177, ACM, December 1981.
Published as *Operating Systems Review* 15(5).

[RB82] L.A. Rowe and K.P. Birman.
A local network based on the UNIX operating system.
IEEE Transactions on Software Engineering, SE-8(2):137–146, March 1982.

[RDH*80] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell.
Pilot: an operating system for a personal computer.
Communications of the ACM, 23(2):81–92, February 1980.
Presented at the 7th Symposium on Operating Systems Principles, ACM, December 1979.

[RR81] R.F. Rashid and G.G. Robertson.
Accent: a communication oriented network operating system kernel.
In *Proc. 8th Symposium on Operating Systems Principles*, pages 64–75, ACM, December 1981.
Published as *Operating Systems Review* 15(5).

[RS83] Jeffrey S. Rosenschein and Vineet Singh.
The Utility of Meta-level Effort.
Report No. HPP-83-20, Heuristic Programming Project, Stanford University, March 1983.

[Sal79] J.H. Saltzer.
Naming and Binding of Objects, chapter 3, pages 99–208.
Springer-Verlag, 1979.

[SBK77] H. Sullivan, T.R. Bashkow, and D. Klappholz.
A large scale homogeneous, fully distributed parallel machine ii.
In *Proceedings of the 4th Annual IEEE Symposium on Computer Architecture*, 1977.

[SF79] M. H. Solomon and R.A. Finkel.
The roscoe distributed operating system.
In *Proc. Symposium on Operating System Principles*, ACM, December 1979.

[SF80] W.D. Sincoskie and D.J. Farber.
The series/1 distributed operating system.
In *Proc. 7th Symposium on Operating Systems Principles*, pages 579–584, ACM, December 1980.

[SG85] V. Singh and M.R. Genesereth.
A variable supply model for distributed deductions.
1985.
Submitted for publication.

[SH80] J.F. Shoch and J.A. Hupp.
Notes on the Worm programs - some early experience with a distributed computation.
Technical Report SSL-80-3, Xerox PARC, 1980.

[Shi83] A. Shiloh.
Load Sharing in a distributed operating system.
Department of Computer Science Tech. Report DCL-TR 83-20, The Hebrew University of Jerusalem, July 1983.

[Smi79] R.G. Smith.
The contract net protocol.
In *Proc. First International Conference on Distributed Computing Systems,* IEEE, 1979.

[Smi80] R.G. Smith.
The contract net protocol: high level communication and control in a distributed problem solver.
***IEEE Transactions on Computers,* C(29):1104-1113, 1980.**

[Sym82] ***Lisp Machine manual***
Symbolics Inc., 1982.

[Tei84] W. Teitelman.
The Cedar Programming Environment: A Midterm Report and Examination.
Technical Report CSL-83-11, Xerox PARC, 1984.

[TM81] A.S. Tannenbaum and S.J. Mullender.
An overview of the Amoeba distributed operating system.
***SIGOPS Operating Systems Review,* 15(l):51-64, July 1981.**

[Uni83] ***UNIX Programmer's Manual.***
University of California at Berkeley, 1983.

[vTW81] A.M. van Tilborg and L.D. Wittie.
Distributed task force scheduling in multi-computer networks.
In *Proceedings of AFIPS,* 1981.

[Wal80] D.W. Wall.
Mechanisms for Broadcast and Selective Broadcast.
PhD thesis, Stanford University, June 1980.
Also available as Computer Science Tech. Report 190.

[WM85] Y. Wang and R.J.T. Morris.
Load sharing in distributed systems.
***IEEE Transactions on Computers,* C-34(3):204-217, March 1985.**

[WN80] M.V. Wilkes and R.M. Needham.
The Cambridge model distributed system.

Operating Systems Review, 14(1):21–29, January 1980.

[WPE*83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel.
The LOCUS distributed operating system.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 49–70, ACM, October 1983.
Published as *Operating Systems Review* 17(5).

[WvT80] L.D. Wittie and A.M. van Tilborg.
Micros, a distributed operating system for micronet, a reconfigurable network computer.
IEEE Transactions on Computers, C(29):1133–1144, December 1980.

[Zwa84] W. Zwaenepoel.
Message Passing on a Local Network.
PhD thesis, Stanford University, September 1984.