The Sprite Network Operating System

John K. Ousterhout
Andrew R. Cherenson
Frederick Douglis
Michael N. Nelson
Brent B. Welch

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

Abstract

Sprite is a new operating system for networked uniprocessor and multiprocessor workstations with large physical memories. It implements a set of kernel calls much like those of 4.3 BSD UNIX, with extensions to allow processes on the same workstation to share memory and to allow processes to migrate between workstations. The implementation of the Sprite kernel contains several interesting features, including a remote procedure call facility for communication between kernels, the use of *prefix tables* to implement a single file name space and to provide flexibility in administering the network file system, and large variable-size file caches on both client and server machines, which provide high performance even for diskless workstations.

Keywords: Operating systems, networks, file systems, process migration, remote procedure call.

1. Introduction

Sprite is an experimental network operating system under development at the University of California at Berkeley. It is part of a larger research project called SPUR (''Symbolic Processing Using RISCs''), whose goal is to design and construct a high-performance multiprocessor workstation with special hardware support for Lisp applications [4]. Although one of Sprite's primary goals is to support applications running on SPUR workstations, we hope that the system will work well for a variety of high-performance engineering workstations. Sprite is currently being used on Sun-2 and Sun-3 workstations.

The motivation for building a new operating system came from three general trends in computer technology: networks, large memories, and multiprocessors. In more and more research and engineering organizations, computing occurs on personal workstations connected by local-area networks, with larger time-shared machines used only for those applications that cannot achieve acceptable performance on workstations. Unfortunately, workstation environments tend to suffer from poor performance and difficulties of sharing and administration, due to the distributed nature of the systems. In Sprite, we hope to hide the distribution as much as possible and make available the same ease of sharing and communication that is possible on time-shared machines.

The second technology trend that drove the Sprite design is the availability of ever-larger physical memories. Today's engineering workstations typically contain 4 to 32 Mbytes of physical memory, and we expect memories of 100-500 Mbytes to be commonplace within a few years. We believe that such large memories will change the traditional balance between computation and I/O by permitting all of users' commonly-accessed files to reside in main memory. The ''RAMdisks'' available on many commercial personal computers have already demonstrated this effect on a small scale. One of our goals for Sprite is to manage physical memory in a way that maximizes the potential for file caching.

The third driving force behind Sprite is the imminent arrival of multiprocessor workstations. Workstations with more than one processor are currently under development in several research organizations (UCB's SPUR, DEC's Firefly, and Xerox's Dragon are a few prominent examples) and we expect multiprocessor workstations to be available from several major manufacturers within a few years. We hope that Sprite will facilitate the development of multiprocessor applications, and that the operating system itself will be able to take advantage of multiple processors in providing system services.

Our overall goal for Sprite is to provide simple and efficient mechanisms that capitalize on the three technology trends. This paper is a discussion of the ways that the trends affected the design of Sprite. In areas where the technology factors did not suggest special techniques, we modelled the system as closely as possible after Berkeley UNIX.

The technology trends had only a minor impact on the facilities that Sprite provides to application programs. For the most part, Sprite's kernel calls are similar to those provided by the 4.3 BSD version of the UNIX operating system. However, we added three additional facilities to Sprite in order to encourage resource sharing: a transparent network file system; a simple mechanism for sharing writable memory between processes on a single workstation; and a mechanism for migrating processes between workstations in order to take advantage of idle machines.

Alhthough the technology trends did not have a large effect on Sprite's kernel interface, they suggested dramatic changes in the kernel implementation, relative to UNIX. This is not surprising, since networks, large memories, and multiprocessors were not important issues in the early 1970's when the UNIX kernel was designed. We built the Sprite kernel from scratch, rather than modifying an existing UNIX kernel. Some of the interesting features of the kernel implementation are:

- The kernel contains a remote procedure call (RPC) facility that allows the kernel of each workstation to invoke operations on other workstations. The RPC mechanism is used extensively in Sprite to implement other features, such as the network file system and process migration.
- Although the Sprite file system is implemented as a collection of *domains* on different server machines, it appears to users as a single hierarchy that is shared by all the workstations. Sprite uses a simple mechanism called *prefix tables* to manage the name space; these dynamic structures facilitate system administration and reconfiguration.

- To achieve high performance in the file system, and also to capitalize on large physical memories, Sprite caches file data both on server machines and client machines. A simple cache consistency mechanism guarantees that applications running on different workstations always use the most up-to-date versions of files, in exactly the same fashion as if the applications were executing on a single machine.
- The virtual memory system uses ordinary files for backing storage; this simplifies the implementation, facilitates process migration, and may even improve performance relative to schemes based on a special-purpose swap area. Sprite retains the code segments for programs in main memory even after the programs complete, in order to allow quick start-up when programs are reused. Lastly, the virtual memory system negotiates with the file system over physical memory usage, in order to permit the file cache to be as large as possible without degrading virtual memory performance.
- Sprite guarantees that processes behave the same whether migrated or not. This is achieved by designating a *home* machine for each process and forwarding location-dependent kernel calls to the process's home machine.

The rest of this paper describes the Sprite facilities and implementation in more detail. Section 2 discusses the unusual features of Sprite's application interface, and Sections 3-7 provide more details on the aspects of the kernel implementation summarized above. Section 8 gives a brief status report and conclusions.

2. Application Interface

Sprite's application interface contains little that is new. The Sprite kernel calls are very similar to those provided by the Berkeley versions of UNIX (we have ported a large number of traditional UNIX applications to Sprite with relatively little effort). This section describes three unusual aspects of the application interface, all of which can be summed up in one word: sharing. First, the Sprite file system allows all of the disk storage and all of the I/O devices in the network to be shared by all processes so that processes need not worry about machine boundaries. Second, the virtual memory mechanism allows physical memory to be shared between processes on the same workstation so that they can extract the highest possible performance from multiprocessors. Third, Sprite implements process migration, which allows jobs to be offloaded to idle workstations and thereby allows processing power to be shared.

2.1. The File System

Almost all modern network file systems, including Sprite's, have the same ultimate goal: *network transparency*. Network transparency means that users should be able to manipulate files in the same ways they did under time-sharing on a single machine; the distridbuted nature of the file system and the techniques used to access remote files should be invisible to users under normal conditions. The LOCUS system was one of the first to make transparency an explicit goal [10]; other file systems with varying degrees of transparency are CMU-ITC's Andrew [5] and Sun's NFS [11].

Most network file systems fail to meet the transparency goal in one or more ways. The earliest systems (and even some later systems, such as 4.2 BSD) allowed remote file access only with a few special programs (e.g., **rcp** in 4.2 BSD); most application programs could only access files stored on local disks. Second-generation systems, such as Apollo's Aegis [6], allow any application to access files on any machine in the network but special names must be used for remote files (e.g., ‘‘file’’ for a local file but ‘‘[ivy]file’’ for a file stored on the Ivy server). Third-generation network file systems, such as LOCUS, Andrew, NFS, and Sprite, provide *name transparency*: the location of a file is not indicated directly by its name, and it is possible to move groups of files from one machine to another without changing their names.

Most third-generation systems still have some non-transparent aspects. For example, in Andrew and NFS only a portion of the file system hierarchy is shared; each machine must also have a private partition that is accessible only to that machine. In addition, Andrew and NFS do not permit applications running on one machine to access I/O devices on other machines. LOCUS appears to be alone among current systems in providing complete file transparency.

Sprite is similar to LOCUS in that it provides complete transparency, so that the behavior seen by applications running on different workstations is exactly the same as it would be if all the applications were executing on a single time-shared machine. There is a single file hierarchy that is accessible uniformly to all workstations. Although it is possible to determine where a file is stored, that information is not needed in normal operation. There are no special programs for operating on remote files as opposed to local ones, and there are no operations that can be used only on local files. Sprite also provides transparent access to remote I/O devices. As in UNIX, Sprite represents devices as special files; unlike most versions of UNIX, Sprite allows any process to access any device, regardless of the device's location.

2.2. Shared Address Spaces

The early versions of UNIX did not permit memory to be shared between user processes, except for read-only code. Each process had private data and stack segments, as shown in Figure 1. Since then, extensions to allow read-write memory sharing have been implemented or proposed for several versions of UNIX, including System V, SunOS, Berkeley UNIX, and Mach.

There are two reasons for providing shared memory. First, the most natural way to program many applications is to use a collection of processes in a shared address space. It is particularly convenient to use multiple processes when an application consists of mostly-independent sub-activities (e.g., one process to respond to keystrokes and another to respond to packets arriving over a network), while the shared address space allows them to cooperate to achieve a common goal (e.g., managing a collection of windows on a screen). The second motivation for shared memory is the advent of multiprocessors. If an application is to be decomposed into pieces that can be executed concurrently, there must be fast communication between the pieces. The faster the communication, the greater the degree of concurrency that can be achieved. Shared memory provides the fastest possible communication, hence the greatest opportunity for concurrent execution.

Sprite provides a particularly simple form of memory sharing: when a process invokes the **Proc_Fork** kernel call to create a new process, it may request that the new process share the parent's data segment (see Figure 2). The stack segment is still private to each process; it contains procedure invocation records and private data for the process. For simplicity, Sprite's mechanism provides all-or-nothing sharing: it isn't possible for a process to share part of its data segment with one process and part with another.

We expect multiprocess applications to synchronize using hardware mutual-exclusion instructions (e.g., test-and-set) directly on shared memory. In most cases it will not be necessary to invoke the kernel, so synchronization can be accomplished in just a few instructions. The kernel participates only when it is necessary to delay the execution of a process, e.g., to wait for a lock to be released. For these situations, Sprite provides kernel calls to put a process to sleep and wake it up again later.

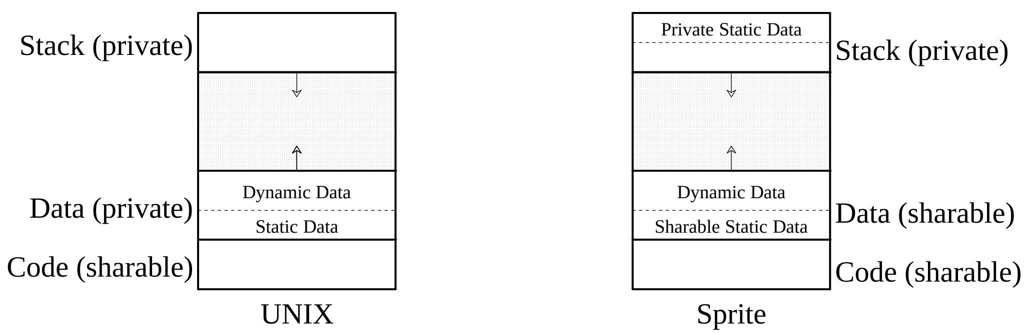


Figure 1. The organization of virtual memory as seen by user processes in traditional UNIX (left) and Sprite (right). In both systems there are three distinct segments. The lower portion of the data segment contains static data known at compile time, and the upper portion expands to accomodate dynamically-allocated data. In UNIX, processes may share code but not data or stack. In Sprite the data segment may be shared between processes, including both statically-allocated and dynamic data. Private static data may be stored at the top of the stack segment.

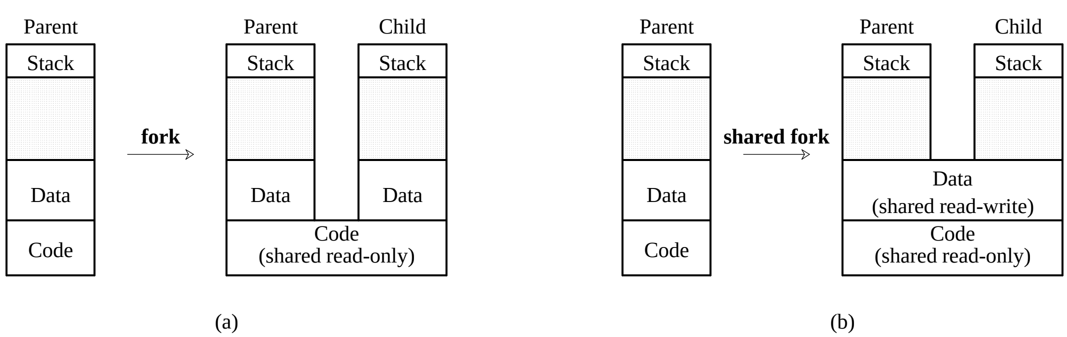


Figure 2. The UNIX **fork** operation, shown in (a), creates a new process that shares code with its parent while using private copies of the data and stack segments. Sprite provides both the traditional **fork** and also a **shared fork**, shown in (b), in which the child shares its parent's data as well as code.

This has allowed us to implement efficient synchronization primitives.

2.3. Process Migration

In an environment with a workstation for each person, many of the machines will be idle at any given time. In order to allow users to harness this idle computing power, Sprite provides a new kernel call, **Proc_Migrate**, which will move a process or group of processes to an idle machine. Processes sharing a heap segment must migrate together. Sprite keeps track of which machines are idle and selects one as the target for the migration. The fact that a process has migrated is transparent both to the migrated process and to the user, as described below. The only noticeable difference after migration will be a reduction in the load of the home machine.

Initially, we expect migration to be used in two ways. First, there are shell commands for manual migration, which allow users to migrate processes in much the same way that the UNIX shell allows users to place processes in background. Second, we have written a new version of the UNIX **make** utility, called **pmake**. **Pmake**, like **make**, carries out the recompilation of programs when their source files change. Whereas **make** invokes the recompilations sequentially, **pmake** is organized to invoke multiple recompilations concurrently, using process migration to offload the compilations to idle machines. We hope to see more and more automatic uses of migration, like **pmake**, in the future.

The idea of moving work to idle machines is not a new one. Unfortunately, though, the most widely available facilities (e.g., the **rsh** command of 4.2BSD UNIX and the **rex** facility of Sun's UNIX) provide only *remote invocation*: the ability to initiate new processes on other machines, but not the ability to move processes once they have started execution. Process migration, which allows processes to be moved at any time, has been implemented in several systems (e.g., LOCUS [10], V [12], and Accent [15]) but is still not widely available. For Sprite we decided to implement process migration. We think that the additional flexibility provided by migration is particularly important in a workstation environment. For example, if remote invocation is used to offload work onto an idle machine and then the machine's user returns, either the foreign processes will have to be killed or else the machine's user will receive degraded response until the foreign processes complete. In Sprite, the foreign processes can be migrated away.

One of the most important attributes of Sprite's migration mechanism is its transparency, both to the process and to the user. This means that a process will produce exactly the same results when migrated as it would if it were not migrated: Sprite preserves the environment of the process as it migrates, including files, working directory, device access, environment variables, and anything else that could affect the execution of the process. In addition, a migrated process appears to its user still to be running on the user's home machine: it will appear in listings of processes on that machine and can be stopped or killed or debugged just like the user's other processes. In contrast, **rsh** does not preserve the working directory and other aspects of the environment, and neither **rsh** nor **rex** allows a remotely-executing process to be examined or manipulated in the same fashion as local processes. Even the other implementations of process migration tend not to provide complete transparency to users, although they do provide complete transparency to the migrated processes. Section 7 describes how Sprite achieves execution transparency.

3. Basic Kernel Structure

Application programs invoke kernel functions via a collection of *kernel calls*. The basic flow of control in a kernel call is similar in Sprite to what it is in UNIX: user processes execute ‘‘trap’’ instructions to switch to supervisor state, and the kernel executes as a privileged extension of the user process, using a small per-process kernel stack for procedure invocation within the kernel.

This section describes two features of Sprite's basic kernel structure that were provided in order to support multiprocessor and network operation. First, a multi-threaded synchronization structure allows the Sprite kernel to run efficiently on multiprocessors. Second, a remote procedure call facility allows kernels to invoke operations remotely over the network.

3.1. Multi-threading

Many operating system kernels, including UNIX, are *single-threaded*: a single lock is acquired when a process calls the kernel and released when the process puts itself to sleep or returns to user state. In these systems processes are never pre-empted while executing kernel code, except by interrupt routines. The single-threaded approach simplifies the implementation of the kernel by eliminating many potential synchronization problems between processes. Unfortunately, though, it does not adapt well to a multiprocessor environment: with more than a few processors, contention for the single kernel lock will limit the performance of the system.

In contrast, the Sprite kernel is *multi-threaded*: several processes may execute in the kernel at the same time. The kernel is organized in a monitor-like style, with many small locks protecting individual modules or data structures instead of a single overall lock. Many processes may execute in the kernel simultaneously as long as they don't attempt to access the same monitored code or data. The multi-threaded approach allows Sprite to run more efficiently on multiprocessors, but the multiplicity of locks makes the kernel more complex and slightly less efficient since many locks may have to be acquired and released over the lifetime of each kernel call.

3.2. Remote Procedure Calls

In designing Sprite for a network of workstations, one of our most important goals was to provide a simple and efficient way for the kernels of different workstations to invoke each others' services. The mechanism we chose is a kernel-to-kernel remote procedure call (RPC) facility similar to the one described by Birrell and Nelson [2]. We chose RPC rather than a message style because RPC provides a simple programming model (remote operations appear just like local procedure calls) and because the RPC approach is particularly efficient for request-response transactions, which we expected to be the most common form of interaction between kernels.

The implementation of RPC consists of *stubs* and *RPC transport*, as shown in Figure 3. Together they hide the fact that the calling procedure and the called procedure are on different machines. For each remote call there are two stubs, one on the client workstation and one on the server. On the client, the stub copies its arguments into a request message and returns values from a result message, so that the calling procedure isn't aware of the underlying message communication. The server stub passes arguments from the incoming message to the desired procedure, and packages

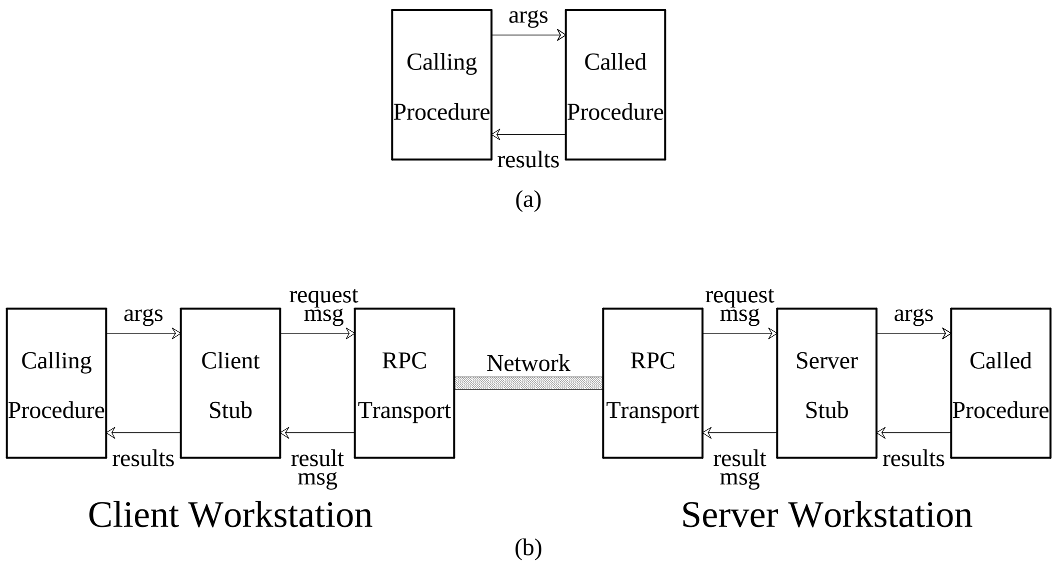


Figure 3. Sprite's remote procedure call mechanism makes it appear as if a remote procedure can be invoked directly, as in (a). The actual situation is shown in (b), where stub procedures copy procedure arguments and results into and out of messages, and a transport mechanism delivers the messages reliably and assigns server processes to requests.

up results from the procedure, so that the called procedure isn't aware that its real caller is on a different machine. Birrell and Nelson modified their compiler to generate the stubs automatically from a specification of the procedures' interfaces. In order to avoid making changes to our C compiler, we hand-generated the stubs for the 40 or so remote operations used in the Sprite kernel. Although this was workable, it would have been more convenient if an automated stub-generator had been available.

The second part of the RPC implementation is RPC transport: it delivers messages across the network and assigns incoming requests to kernel processes that execute the server stubs and called procedures. The goal of RPC transport is to provide the most efficient possible communication between the stubs while ensuring that messages are delivered reliably. Sprite's RPC transport uses two techiques to gain efficiency: implicit acknowledgements and fragmentation.

Since network transmission is not perfectly reliable, each request and response message must be acknowledged; if no acknowledgment is received within a ''reasonable'' time, the sender retransmits. To reduce the overheads associated with processing acknowledgement packets, Sprite uses the scheme described by Birrell and Nelson, where each request or response message serves as an implicit acknowledgement for the previous response or request message from that client, respectively. In the common case of short closely-spaced operations, only two packets are transmitted for each remote call: one for the request and one for the response.

The simplest way to implement RPC is to limit the total size of the arguments or results for any given RPC so that each request and response message can fit into a single network packet. Unfortunately, the maximum allowable size for a network packet is relatively small (about 1500 bytes for Ethernet), so this approach would result in high overhead for bulk transfers: the delays associated with sending a request, dispatching to a server process, and returning a response would be incurred for each 1500 bytes. One of the most common uses of RPC is for remote file access, so we were unwilling to accept this performance limitation.

Sprite's RPC mechanism differs from the Birrell-Nelson scheme in that it uses fragmentation to ship large blocks of data (up to 16 Kbytes) in a single remote operation. If a request or reply message is too long to fit in a single packet, RPC transport breaks the message up into multiple packets (''fragments'') which it transmits in order without waiting for acknowledgment. The receiving RPC transport reassembles the fragments into a single large message. A single acknowledgement is used for all the fragments, using the same implicit acknowledgement scheme described above. When packets are lost in transmission, the acknowledgment indicates which fragments have been received so that only the lost fragments are retransmitted.

Sprite kernels trust each other, and we assume that the network wire is physically secure (all workstations on the network must run the Sprite kernel or some other trustworthy software). Thus the RPC mechanism does not use encryption, nor do the kernels validate RPC operations except to prevent user errors and detect system bugs. The RPC mechanism is used only by the kernels, and is not directly visible to user applications.

Figure 4 shows the measured performance of the Sprite RPC mechanism. Figure 4(a) shows that the minimum round-trip time for the simplest possible RPC is about 2.8 ms between Sun-3/75 workstations, with an additional 1.2 ms for each Kbyte of data. Figure 4(b) shows that throughputs greater than 700 Kbytes/sec (nearly 60% of the total Ethernet bandwidth of 10 Mbits/sec) can be achieved

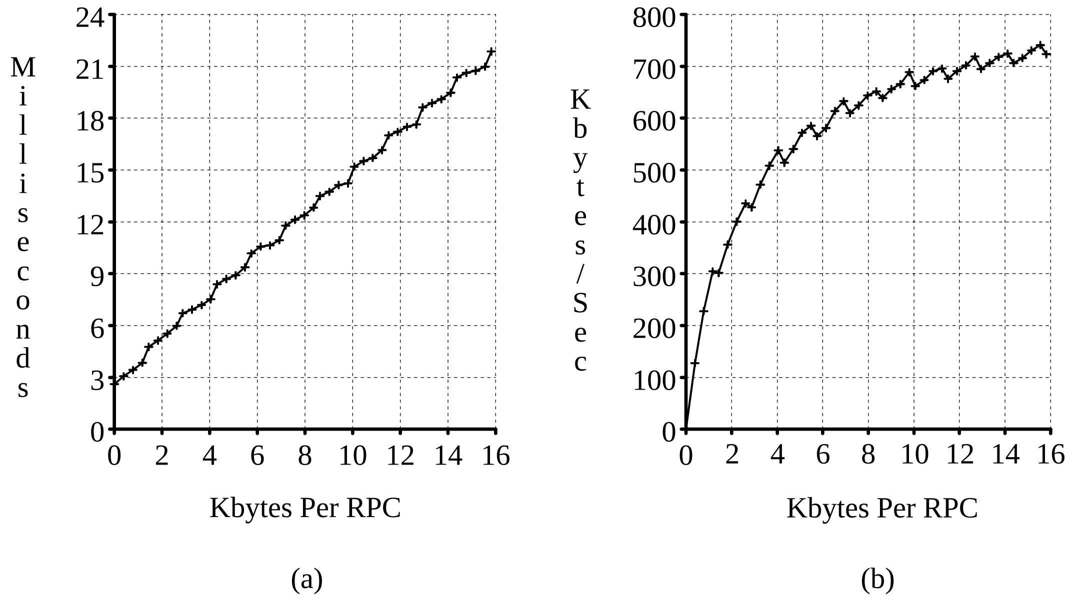


Figure 4. Measured performance of Sprite's RPC mechanism between Sun-3/75 workstations. The test consisted of one kernel invoking a remote procedure in another kernel, passing it the contents of a variable-size array as an argument. The called procedure returned immediately. (a) shows the round-trip time for an individual RPC as a function of the amount of data passed to the remote procedure, and (b) shows the throughput when repeated RPCs are made. Larger transfers, which use fragmentation on 1500-byte boundaries, are most efficient. The jumps in the curves occur at the points where additional packets become necessary.

between two workstations if each RPC transfers a large amount of data. Without fragmentation (at most 1500 bytes transmitted per RPC) the throughput is reduced by more than a factor of two. The measurements in Figure 4 are for operations between kernels. User-visible performance is slightly worse: for example, a user process can achieve a throughput of only 475 Kbytes/sec when it reads a file that is cached in the main memory of a remote server and the kernel makes 4-Kbyte RPC requests.

4. Managing the File Name Space – Prefix Tables

In designing the Sprite file system for a network environment, we were particularly concerned about two implementation issues: how to manage the file name space in a way that simplifies system administration, and how to manage the file data in a way that provides high performance. Furthermore, we felt that it was important to provide easy administration and high performance without compromising users' ability to share files. This section discusses the naming issue, and Section 5 discusses the performance issue.

To users, the Sprite file system is a single hierarchy, just as in time-shared UNIX. To system administrators, the file system is a collection of *domains*, which are similar to ‘‘file sytems’’ in UNIX.

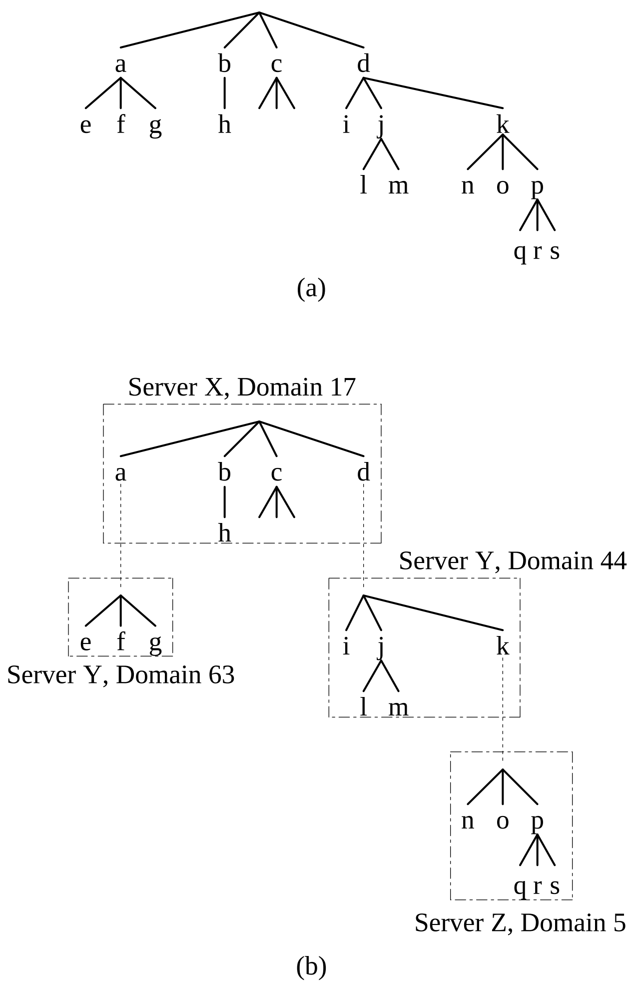


Figure 5. Although the Sprite file system behaves as if it were a single hierarchy, as shown in (a), it is actually divided up into domains, as shown in (b). Each domain may be stored on a different server.

Prefix	Server	Token
/	X	17
/a/	Y	63
/d/	Y	44
/d/k/	Z	5

Table 1. A prefix table corresponding to the domain structure of Figure 5. Prefix tables are loaded dynamically, so they need not hold complete file information at any given time.

Each domain contains a tree-structured portion of the overall hierarchy. The domains are joined into a single hierarchy by overlaying the leaves of some domains with the roots of other domains as illustrated in Figure 5. (In UNIX terms, the sub-domains are *mounted* on their parents; the leaves where mounting occurs, such as ''/a'' in Figure 5, are called *mount points*) As the operating system traverses the components of a file name during name lookup, it must move automatically from domain to domain in order to keep the domain boundaries from being visible to users.

The interesting naming issues are how to keep track of the domain structure and how to handle file names that cross domain boundaries. These issues are particularly interesting in a network environment where the domains may be stored on different servers and where the server configuration may change frequently. UNIX and most of its derivatives (such as NFS) use static *mount tables* to keep track of domains; the mount tables are established by reading a local configuration file at boot-time. This makes it difficult for the systems to respond to configuration changes. In our NFS clusters, for example, any change to the domain structure typically requires each user to modify the configuration file on his or her workstation and reboot. Even in small clusters we have found that such changes occur distressingly often.

In Sprite, we use a more dynamic approach to managing the domain structure, which we call *prefix tables*. The kernel of each client machine maintains a private prefix table. Each entry in a prefix table corresponds to a domain; it gives the full name of the top-level directory in the domain (i.e., the common prefix shared by the names of all files in the domain), the name of the server on which that domain is stored, and an additional token to pass to the server to identify the domain (see Table 1). Prefix tables are not normally visible to user processes. The sections below describe how prefix tables are used during file name lookup and how they are created using a dynamic broadcast protocol.

4.1. Locating a File

In Sprite, as in UNIX, application programs refer to files by giving either an *absolute path name* for the file (one starting at the file system root, such as ''/d/k/p/r'' in Figure 5), or a *relative path name*, which is interpreted starting at a previously-specified *working directory* (if the working directory is

‘‘/d/k/p’’ in Figure 5, then the relative name ‘‘r’’ refers to the same file as ‘‘/d/k/p/r’’). To lookup an absolute path name, a client kernel matches the name against all of the entries in its prefix table and chooses the entry with the longest matching prefix. In the example of Figure 5, the file name ‘‘/d/k/p/r’’ will match three entries in the table, of which the entry for server Z has the longest prefix. The client strips the prefix from the file name and uses the RPC facility to send the remainder of the name (‘‘p/r’’) to the server, along with the token from the prefix table entry (5). The server uses the token to locate the root directory of the domain, looks up the remainder of the file name, and replies with a token identifying the file. The client can then issue read, write, and close requests by making RPCs to the server with the file’s token.

Sprite handles working directories by opening the working directory and storing its token and server address as part of the process’ state. When a file name is specified relative to the working directory, the client kernel uses the token and server address corresponding to the working directory rather than those from a prefix table entry. Thus absolute and relative path name lookups appear identical to the server.

There are several cases where the initial server that receives a file name cannot completely process the name. These correspond to situations where the file’s name crosses a domain boundary. For example, ‘‘..’’ components in a name (which refer to the parent directory) could cause it to ascend back up the hierarchy and out the top of the domain, or the name could refer to a symbolic link containing an absolute file name for a different domain, or a relative path name could start at the current working directory and descend down into a new domain. In each of these cases, the initial server processes as many components of the file name as it can, then returns a new name to the client instead of a file token. The client takes the new name, processes it with its prefix table, and sends it to a new server. This process repeats until the name is completely resolved (see [14] for details).

The prefix approach bypasses the root domain (and its server) when looking up absolute names of files in non-root domains. Since a large fraction of all name lookups involve absolute path names, we expect this approach to reduce the load on the root server and increase the scalability of the system relative to schemes that require root server participation for every absolute path name. It may also permit the system to provide limited service even when the root server is down.

4.2. Managing Prefix Tables

One of the greatest advantages of prefix tables is that they are created dynamically and updated automatically when the system configuration changes. To add a new entry to its prefix table, a client broadcasts a prefix name to all servers. The server storing the domain replies with its address and the token corresponding to the domain. The client uses this information to create a new prefix table entry. Initially, each client starts out with an empty prefix table and broadcasts to find the entry for ‘‘/’’. As it uses more files, it gradually adds entries to its prefix table.

How does a client know when to add a new prefix to its table? The file at the mount point for each domain is a special link called a *remote link*, which identifies the file as the mount point for a new domain. For example, in Figure 5 the file ‘‘/d/k’’ in Server Y’s domain is a remote link. A remote link is similar to a symbolic link in that it stores a file name; for remote links, this is the prefix name (i.e., the file’s absolute name). Whenever a remote link is encountered in file name lookup, the server returns to the client the prefix name and the remainder of the name being looked up. The client uses the broadcast protocol to make a new prefix table entry and then reprocesses the remainder of the name. Remote links do not store any network address information; they simply indicate the presence of a domain somewhere. This feature permits the system to adapt quickly to changes in configuration.

Prefix table entries are treated as hints and are adjusted automatically as the system configuration changes. When a client sends an open request to a server, it is possible for the request to fail with a timeout (if the server has crashed) or with a rejection (if the server no longer stores the domain). In either case, the client invalidates the prefix table entry for the domain and rebroadcasts. If the domain has moved, the new server will respond to the rebroadcast and the client will establish a new prefix table entry and retry the open; in this case the change of configuration will be invisible to user processes. If the server has crashed, then the broadcast will timeout; each additional open will also broadcast and timeout. During the time the server is down, user processes will receive errors analogous to ‘‘disk off-line’’ errors in time-shared UNIX. Eventually the domain will become available again, and the next open will re-establish the prefix table entry. To add a new domain to the file system, all that is needed is to add a remote link at the mount point for the domain and arrange for the server to respond to requests.

5. Managing File Data – Client and Server Caches

The Sprite file system is implemented using large caches of recently-used file blocks stored in the main memories of both clients and servers. The caches provide two benefits, which are especially important when most of the workstations are diskless. First, the caches improve file system performance by eliminating disk accesses and network transactions. Second, they reduce the loading on the network and the servers, which increases the scalability of the system. Sprite's caches use a consistency protocol that allows applications on different workstations to share files just as if they were running on a single time-sharing system.

5.1. Basic Cache Design

Each client and server workstation maintains a large cache of recently-accessed file blocks, as shown in Figure 6. The caches are organized on a block basis, rather than a whole-file basis as in the Andrew file system [5], and are stored in main memory rather than on a local disk. Blocks are currently 4 Kbytes. Each block in the cache is identified by a token for a file and a block location within the file. When the **Fs_Read** kernel call is invoked to read a block of a file, the kernel first checks its cache and returns the information from the cache if it is present. If the block is not in the cache, then the kernel reads it from disk (if the file is on a local disk), or requests it from a server; in either case, the block is added to the cache, replacing the least-recently-used block. If the block is requested from a server, the server checks its own cache before issuing a disk I/O and adds the block to its cache if the block was not there already.

File writes are handled in Sprite using a *delayed-write* approach. When an application issues an **Fs_Write** kernel call, the kernel simply writes the block into its cache and returns to the application. The block is not written through to the disk or server until either it is ejected from the cache or 30

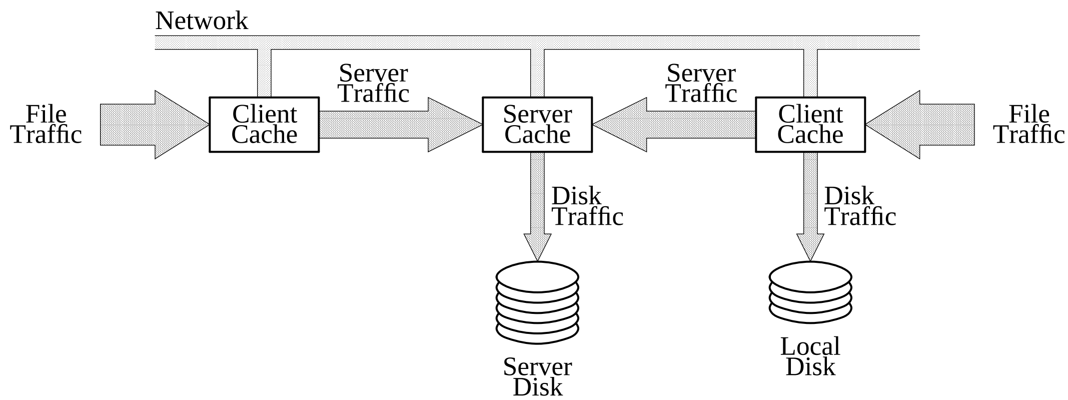


Figure 6. Caches in the Sprite file system. When a process makes a file access, it is presented first to the cache of the process's workstation (''file traffic''). If not satisfied there, the request is passed either to a local disk, if the file is stored locally (''disk traffic''), or to the server where the file is stored (''server traffic''). Servers also maintain caches in order to reduce their disk traffic.

seconds have elapsed since the block was last modified. This policy is similar to the one used in time-shared UNIX; it means some recent work may be lost in a system crash, but it provides much higher performance to applications than a policy based on write-through, since the application can continue without waiting for information to be flushed to disk. For applications with special reliability requirements, Sprite provides a kernel call to flush one or more blocks of a file to disk.

5.2. Cache Consistency

When clients cache files, a consistency problem arises: what happens if one client modifies a file that is cached by other clients? Can subsequent references to the file by the other clients return ''stale'' data? Most network file systems, such as Sun's NFS, provide only limited guarantees about consistency. In NFS, for example, other clients with the file open may see stale data until they close the file and re-open it. Sprite guarantees consistency: each **Fs_Read** kernel call always returns the most up-to-date data for a file, regardless of how the file is being used around the network. This means that application programs will have the same behavior running on different workstations under Sprite as they would if they were all running on a single time-shared UNIX system.

To simplify the implementation of cache consistency, we considered two separate cases. The first case is *sequential write-sharing*, where one workstation modifies a file and another workstation reads it later, but the file is never open on both workstations at the same time. We expect this form of write-sharing to be the most common one. The second case is *concurrent write-sharing,* where one workstation modifies a file while it is open on another workstation. Our solution to this situation is more expensive, but we do not expect it to occur very often.

Sprite handles sequential write-sharing using version numbers. When a client opens a file, the server returns the current version number for the file, which the client compares to the version number associated with its cached blocks for the file. If they are different, the file must have been modified recently on some other workstation, so the client discards all of the cached blocks for the file and reloads its cache from the server when the blocks are needed. The delayed-write policy used by Sprite means that the server doesn't always have the current data for a file (the last writer need not have flushed dirty blocks back to the server when it closed the file). Servers handle this situation by keeping track of the last writer for each file; when a client other than the last writer opens the file, the server forces the last writer to write all its dirty blocks back to the server's cache. This guarantees that the server has up-to-date information for a file whenever a client needs it.

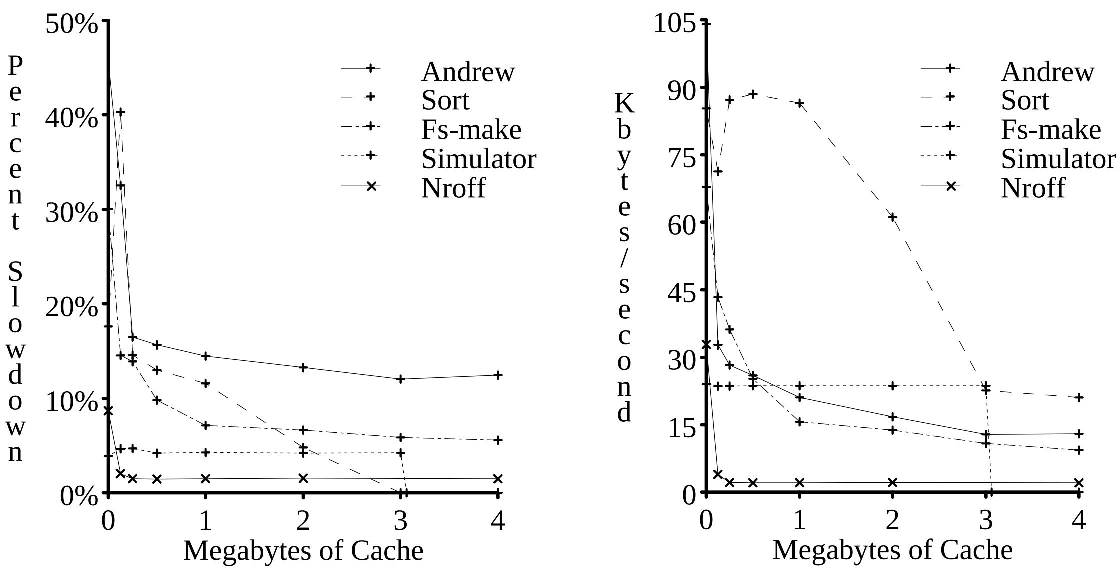
For concurrent write-sharing, where the file is open on two or more workstations and at least one of them is writing the file, Sprite disables client caching for that file. When the server receives an open request that will result in concurrent write-sharing, it flushes dirty blocks back from the current writer (if any), and notifies all of the clients with the file open that they should not cache the file anymore. Cache disabling is done on a file-by-file basis, and only when concurrent write-sharing occurs. A file may be cached simultaneously by several active readers.

There are two potential disadvantages to Sprite's cache consistency mechanism. First, it results in substantially slower file access when caching has been disabled. Fortunately, measurements and simulations in [8] and [9] show that files tend to be open only for short periods of time and are rarely write-shared, so that cache disabling will not occur often. Second, the Sprite approach depends on the fact that the server is notified whenever a file is opened or closed. This prohibits performance optimizations (such as name caching) in which clients open files without contacting the files' servers. Our benchmark results in [8] suggest that such optimizations would only provide small additional performance improvements.

It is important to distinguish between consistency and correct synchronization. Sprite's mechanism provides consistency: each read will return the most up-to-date data. However, the cache consistency mechanism will not guarantee that applications perform their reads and writes in a sensible order. For this, applications must synchronize their actions on the file using the **Fs_Lock** system call or other available communication mechanisms. The cache consistency provided by Sprite simply eliminates the network issues and reduces the problem to what it was on time-sharing systems.

5.3. File System Performance

We ran a series of file-intensive benchmark programs on Sun-3/75 workstations in order to measure the benefits of caching. A single Sun-3/180 file server was used for all I/O and paging traffic



(a) (b)

Figure 7. Client degradation and network traffic as a function of maximum client cache size, for diskless Sun-3/75's with client caches using an unloaded Sun-3/180 file server. For each point the cache size was allowed to vary (as described in Section 6) up to the given maximum. Figure (a) plots degradation, which is the additional time required by a diskless workstation to complete the benchmark, relative to the time to complete the benchmark with a local disk and 4-Mbyte cache. Figure (b) plots network traffic, including bytes transmitted in packet headers and control packets as well as file data. See [8] for a description of the benchmarks.

from the clients. Because the benchmarks do not involve file sharing, they do not measure the overheads associated with cache consistency. This section presents only summary results; for descriptions of the benchmarks and additional performance measurements, see [8].

Figure 7 shows that with caches of a few megabytes, diskless workstations can achieve performance within 1% to 12% of workstations with local disks, whereas without caches the diskless workstations typically run 10% to 40% more slowly than workstations with disks. It also shows that client caching reduces network traffic by a factor of four or more. Without client caching, we believe that the 10-Mbit/sec. bandwidth of Ethernet will be a major bottleneck for next-generation workstations with 5 to 10 MIPS of processing power (e.g., SPUR or the Sun-4 family). Even with client caching, faster networks will be needed to support the next generation of workstations after that. Figure 8 shows that client caching reduces the load on servers by about a factor of two in comparison to systems without client caching, and suggests that a single server can support ten or more active clients

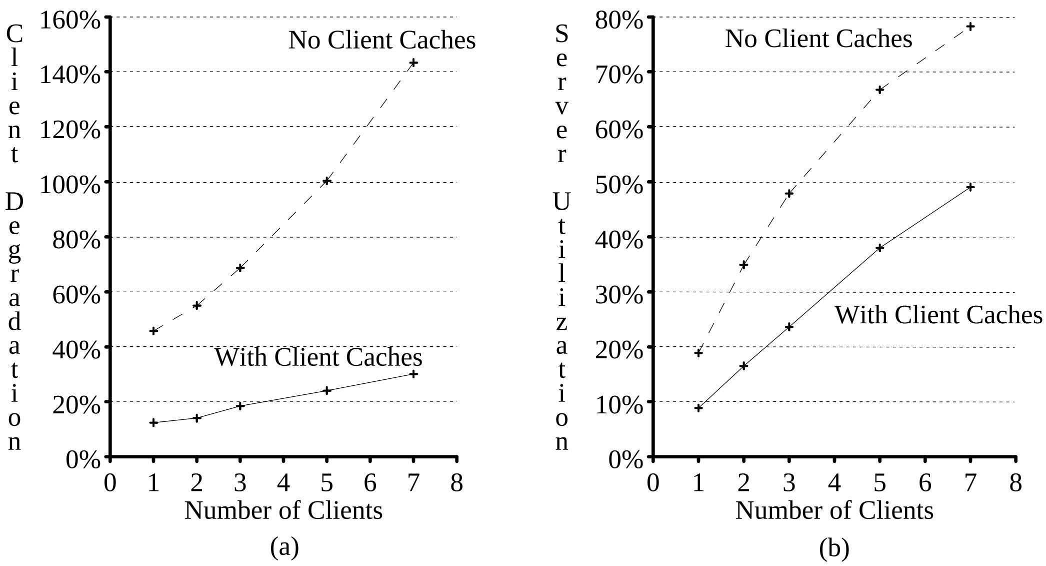


Figure 8. Effects of server contention when multiple diskless clients ran the most intensive benchmark (Andrew) simultaneously on different files using Sun-3/75 workstations. Andrew was written by M. Satyanarayanan [5]; it is a composite benchmark that includes directory searches, file copying, version checking, and compilation. Figure (a) shows additional time required by each diskless client to complete the benchmark, relative to a single client running with local disk and cache. Figure (b) shows server CPU utilization. When client caches were enabled, they were allowed to grow up to 4 Mbytes in size.

without excessive performance degradation. Normal users are rarely as active as the benchmark in Figure 8: references [5] and [8] estimate that one instance of the benchmark presents a load equivalent to at least five average users. This suggests that a Sun-3/180 Sprite file server can support at least 50 user workstations.

In comparisons with Sun's NFS, the Andrew benchmark completed 30% faster under Sprite than under NFS, and generated only about one-fourth of the server load of NFS. Since our NFS servers can support 10-20 clients, the NFS comparison supports our estimate of at least 50 clients per Sprite file server. See [8] for more information on the NFS comparison.

6. Virtual Memory

Sprite's implementation of virtual memory is traditional in many respects. For example, it uses a variation on the ‘‘clock’’ algorithm for its page replacement mechanism, and provides shared read-write data segments using a straightforward extension of the mechanism that provides shared read-only code in time-shared UNIX. These and other aspects of the virtual memory system are described in detail in [7]. This section focusses on three aspects of the virtual memory implementation where we intentionally deviated from UNIX in order to make better use of networks and large physical memories. First, Sprite uses ordinary files for backing storage in order to simplify process migration, to share backing storage between workstations, and to capitalize on server caches. In addition, Sprite provides ‘‘sticky segments’’ and a dynamic tradeoff of physical memory between the virtual memory system and the file cache; these mechanisms were implemented in order to make the best possible use of physical memory as a cache for programs and files.

6.1. Backing Storage

Backing storage is the portion of disk used to hold pages that have been swapped out of physical memory. Most versions of UNIX use a special disk partition for backing storage and manage that partition with special algorithms. In networked UNIX systems, each machine has its own private disk partition for backing storage. In contrast, Sprite uses ordinary files, stored in the network file system,

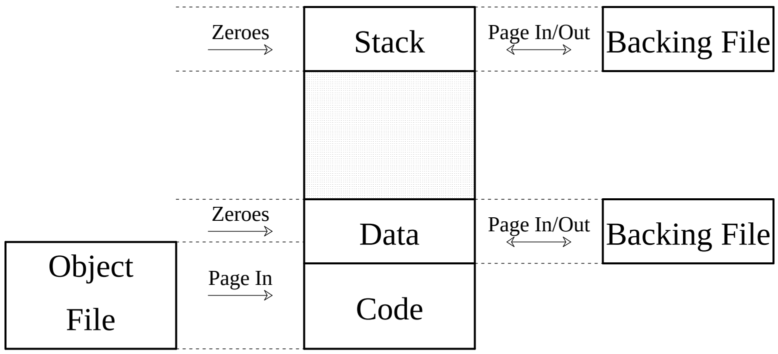


Figure 9. Sprite's paging structure. Code is paged in on-demand from the process's object file; since code is read-only, it need not be written to backing storage and can be re-loaded from the object file when needed. An ordinary file is used to back each of the data and stack segments. Initialized portions of the data segment are read in from the object file on first reference, then written to the backing file during page replacement and re-used from there. For the stack segment and the uninitialized portions of the data segment, pages are filled with zeros on first reference, then paged to and from the backing files.

for backing storage. A separate backing file is used for each data and stack segment, as illustrated in Figure 9. Each workstation is assigned a separate directory in which to create backing files for its processes.

There are several advantages to paging from files. First, it simplifies the implementation of virtual memory by re-using the existing file mechanisms. Second, it provides flexibility not present when each machine uses a private partition for backing storage. Many workstations may store their backing files in the same file system domain; this results in more efficient use of disk space than schemes based on statically-allocated private partitions. The network file system also makes it easy for backing files to be allocated either on local disks or remote servers, and simplifies the implementation of process migration by making all the backing files accessible to all the workstations (see Section 7 below).

Backing files also have interesting performance consequences. In Sprite, remote backing files are cached in the main memories of servers, just like all other files. Our initial measurements show that a client can read random pages from a file in the server's cache faster than from a local disk, which means that a server with a large cache may provide better paging performance than a local disk. We think that CPU and network speeds are likely to increase at a much faster rate than disk speeds over the next few years, which will make remote paging to/from a server's cache even more attractive in the future.

6.2. Sticky Segments

When a program starts execution, the pages in its code and data segments are loaded on-demand from the program's object file when page faults occur. In order to reduce this cost for programs that are invoked frequently, Sprite keeps the code pages of a program in memory even after the program exits. The pages remain in memory until they are replaced using the normal clock mechanism. We call this mechanism *sticky segments*. If the same object file is re-invoked, then the new process can be started more quickly by re-using the sticky segment. If the object file is modified between executions, then the sticky segment will be discarded on the next execution. Data and stack segments are modified during execution, so they cannot be retained after the process completes.

6.3. Double-Caching

Double-caching (caching the same block of a file in two different places in memory) is a potential issue because the virtual memory system is a user of the file system. A naive implementation might cause pages being read from backing files to end up in both the file cache and the virtual memory page pool; pages being eliminated from the virtual memory page pool might simply get moved to the file cache, where they would have to age again before being sent to the server. To avoid these inefficiencies, the virtual memory system bypasses the local file cache when reading and writing backing files. A similar problem occurs when demand-loading code from its executable file. In this case, the pages may already be in the file cache (e.g., because the program was just recompiled). If so, the page is copied to the virtual memory page pool and the block in the file cache is given an infinite age so that it will be replaced before anything else in memory. The sticky segment mechanism will cache the page in the virtual memory system, so it isn't necessary to keep it in the file cache as well. For the portions of object files corresponding to data pages, Sprite permits double-caching in order to provide faster startup of programs (the dirty data pages are discarded on program exit, but clean ones can be quickly re-loaded from the file cache).

Although the virtual memory system bypasses its local file cache when reading and writing backing files, the backing files *will* be cached on servers, as described in Section 6.1 above. This makes servers' memories into an extended main memory for their clients. Servers do not cache backing files for their own processes, since this would constitute double-caching; they only cache backing files for their clients.

6.4. VM-FS Negotiation

The virtual memory system and file system have conflicting needs for physical memory. File system performance is best when the file cache is as large as possible, while virtual memory performance will be best when the file cache is as small as possible so that most of the physical memory may be used for virtual memory. In order to get the best overall performance, Sprite allows the file cache on each workstation to grow and shrink in response to changing demands on the machine's virtual memory and file system. This is accomplished by having the two modules negotiate over physical memory usage. The result is that small I/O-intensive programs, like compilers, may use almost all of memory for a file cache, while large CPU-bound programs may use almost all of memory for their virtual address spaces.

The file system and the virtual memory system manage separate pools of physical memory pages. Each module keeps an approximate time-of-last-access for each page (using different techniques in each module). Whenever either module needs additional memory (because of a page fault or a miss in the file cache), it compares the age of its oldest page with the age of the oldest page from the other module, replacing whichever is older. This allows memory to flow back and forth between the virtual memory page pool and the file cache, depending on the needs of the current applications.

We also considered more centralized approaches to trading off physical memory between the virtual memory page pool and the file cache. One possibility would be to access all information through the virtual memory system. To access a file, it would first be mapped into a process's virtual address space and then read or written just like virtual memory, as in Apollo's Aegis system [6] or Mach[1]. This approach would eliminate the file cache entirely; the standard page replacement mechanisms would automatically balance physical memory usage between file and program information. We rejected the mapped-file approach for several reasons, the most important of which is that it would have forced us to use a more complicated cache consistency scheme. A mapped-file approach requires a file's pages to be cached in a workstation's memory before they can be accessed, so we would not have been able to implement cache consistency by refusing to cache shared files. A second reason for rejecting the mapped-file approach is that we wished to retain the UNIX notion that I/O devices and files are accessed in exactly the same fashion; a mapped-file framework, with the assumed ability to access bytes in random order, does not seem natural for device I/O, which is most often sequential.

7. Process Migration

Sprite's implementation of process migration differs from other implementations such as those in the V System [12], Accent [15], or LOCUS [10] in two major ways. The first difference is the way in which the virtual memory of a process is transferred between machines, and the second difference is the way in which migration is made transparent to the migrated process.

The simplest approach to process migration is:

[1] ‘‘Freeze’’ the process (prevent it from executing any more).

[2] Transfer its state to the new machine, including registers and other execution state, virtual memory, and file access.

[3] ‘‘Unfreeze’’ the process on its new machine, so that it may continue executing.

The virtual memory transfer is the dominant cost in migration, so various techniques have been applied to reduce it. For example, V uses *pre-copying*, where the process continues executing while its memory is transferred; then the process is frozen and any pages that have been modified are re-copied. Accent uses a ‘‘lazy’’ approach to copying, where the virtual memory image is left on the old machine and transferred to the new machine one page at a time when page faults occur. LOCUS checks for a read-only code segment and re-opens it on the new machine, rather than copying it from the old machine; this allows the process to share a pre-existing copy of the code on the new machine, if there is one. In Sprite, backing files simplify the transfer of the virtual memory image. The old machine simply pages out the process's dirty pages and transfers information about the backing files to the target machine. If the code segment already exists on the new machine, the migrating process shares it, as in LOCUS. Pages get reloaded in the process's new machine on demand, using the standard virtual memory mechanisms. Thus, the process need only be frozen long enough to write out its dirty pages. Sprite's approach requires processes to be frozen longer than either V or Accent, but it requires less total data to be copied than V and it does not require the old machine to service page faults after the process is unfrozen on its new machine.

The second, and more important, issue in process migration is how to achieve transparent remote execution. A process must produce the same results when migrated as it would have produced if it had not been migrated, and it must not be necessary for a process to be coded in a special way for it to be migratable. For message-based systems like V and Accent, transparency is achieved by redirecting the process's message traffic to its new home. Since processes communicate with the rest of the world only by sending and receiving messages, this is sufficient to guarantee transparency. In contrast, Sprite processes communicate with the rest of the world by invoking kernel calls. Kernel calls are normally executed on the machine where invoked (unless they make remote procedure calls to other kernels), and some kernel calls will produce different results on different machines. For example, Sprite kernels maintain shared environment variables; **Proc_GetEnviron** may return different results on different machines.

Sprite achieves transparency in a fashion similar to LOCUS by assigning each process a *home node*. A process's home node is the machine on which the process was created, unless the process was created by a migrated process; in this case, the process's home node is the same as the home node of its parent. Whenever a process invokes a kernel call whose results are machine-dependent, the kernel call is forwarded to the process's home node (using the RPC mechanism) and executed there. This guarantees that the process produces the same results as if it were executing at home. To the outside world, the process appears still to be executing at home; its process identifier does not change, the process will appear in a listing of all processes on the home node, and it may be debugged and terminated in the same way as other processes on the home node.

For each kernel call, we thus had two choices: either transfer all the state associated with the call at migration time, so that the call can be executed remotely, or forward home all invocations of the call made by migrated processes. For calls that are invoked frequently, such as all the file system calls, we chose the first course (this was particularly simple for files, since the cache consistency mechanism already takes care of moving the file's data between caches). For infrequently-invoked calls, or those whose state is difficult or impossible to transfer (e.g., calls that deal with the home node's process table) we chose the forwarding approach.

Table 2 gives some preliminary measurements of the costs of migrating a process. If a process is migrated when it starts execution (before it has generated many dirty pages), the migration requires only a few hundred milliseconds on Sun-3/75 workstations. We expect this to be the most common scenario. The other major use of migration will be to evict migrated processes from a workstation whose user has just returned. In this case the major factor will be the number of dirty pages. Even in the worst case (all of memory dirty), all processes can be evicted from an 8-Mbyte workstation in about 15-20 seconds. Table 3 shows that the costs of remote execution are acceptable (less than 5% penalty over executing at home, for a compilation benchmark) and that migration may allow a collection of jobs to be completed much more quickly than they could without migration. See [3] for more information on process migration in Sprite.

Action	Cost or Speed
Migrate smallest possible process	190 msec
Flush dirty pages	585 Kbytes/sec
Demand-load pages	545 Kbytes/sec
Transfer info for open files	14 msec/file
Flush file cache	585 Kbytes/sec

Table 2. The time required to migrate a process on Sun-3/75 workstations. The total time depends on how many dirty pages the process has (these must be flushed to the server during migration), how large its address space is (pages must be loaded on-demand on the process's new host), how many open files it has, and how many dirty blocks for those files are cached locally (they must be flushed). ''Smallest possible process'' refers to a process with no open files and one page each of code, data, and stack.

Program	Execution Time		Improvement
	local	migrated	
One compilation	15.5 sec	15.9 sec	–3%
Two compilations	30 sec	17 sec	43%
Three compilations	45 sec	18 sec	60%
Four compilations	60 sec	20 sec	67%

Table 3. Costs and benefits of process migration, measured by running several compilations concurrently. In the ‘‘local’’ column all the compilations were run concurrently on a single machine. In the ‘‘migrated’’ column one compilation was run locally and each of the others was migrated to a different workstation (except for the ‘‘One compilation’’ row, where the single compilation was migrated).

8. Status and Conclusions

As of this writing, all of the features described in this paper are operational except for the code to choose a target for process migration and to evict migrated processes when a workstation's user returns (this is currently under development). In addition to the features described here, Sprite supports the Internet protocol family (IP/TCP) for communication with other systems, and we plan to support Sun's Network File System protocols in the future. The Sprite kernel contains approximately 100,000 lines of code, of which about half are comments. All but a few hundred lines of code are in C; the remainder are written in assembler. Sprite currently runs on Sun-2 and Sun-3 workstations. We have recently begun to use it for all of our everyday computing, including maintaining Sprite. We plan to port Sprite to the SPUR multiprocessor as prototypes become available in 1988. We hope that Sprite will be portable enough to run on a variety of workstation platforms, and that it will be attractive enough that other people outside the Sprite group will want to use it for their everyday computing.

In conclusion, we hope that Sprite will provide three overall features: sharing, flexibility, and peformance. Sharing is what users want, in order to work cooperatively and make the best use of the hardware resources. Sprite provides sharing at several levels: tightly-coupled processes on the same workstation may share memory; processes everywhere may share files; and users may share processing power using the process migration mechanism. Flexibility is what system administrators want, in order to allow the system to evolve gracefully. Sprite provides flexibility in the form of prefix tables, which allow the file system to reconfigure transparently to users, and in the form of backing files, which allow backing storage to be shared between workstations. Finally, performance is what everyone wants. Sprite provides high performance by using a special-purpose RPC protocol for communication between kernels, and by using physical memory as a flexible cache for both programs and files.

9. Acknowledgments

Adam de Boor implemented the Pmake program and has assisted in many other areas of Sprite development, including porting the X11 window system. David Anderson, Jim Larus, Rita Ousterhout, Gerald Popek, Carlo Séquin, and the *IEEE Computer* referees provided helpful comments on early drafts of this paper. The work described here was supported in part by the Defense Advanced Research Projects Agency under Contract No. N00039-85-R-0269 and in part by the National Science Foundation under grant ECS-8351961.

10. References

[1] Accetta, M., et al. ‘‘Mach: A New Kernel Foundation for UNIX Development.’’ *Proceedings of Summer USENIX*, July 1986, pp. 93-112.

[2] Birrell, A. and Nelson, B. ‘‘Implementing Remote Procedure Calls.’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1986, pp. 39-59.

[3] Douglis, F. and Ousterhout, J. *Process Migration in the Sprite Operating System*. To appear, *7th International Conference on Distributed Computing Systems*, September 1987. Also appears as technical report UCB/CSD 87/343, February 1987.

[4] Hill, M., et al. ‘‘Design Decisions in SPUR.’’ *IEEE Computer*, Vol. 19, No. 11, November 1986, pp. 8-22.

[5] Howard, J., et al. ‘‘Scale and Performance in a Distributed File System.’’ To appear, *ACM Transactions on Computer Systems*.

[6] Leach, P., et al. ‘‘The Architecture of an Integrated Local Network.’’ *IEEE Transactions on Selected Areas in Communications,* Vol. SAC-1, No. 5, November 1983, pp. 842-857.

[7] Nelson, M. *Virtual Memory for the Sprite Operating System.* Techical report UCB/CSD 86/301, June 1986.

[8] Nelson, M., Welch, B., and Ousterhout, J. ‘‘Caching in the Sprite Network File System.’’ To appear, *ACM Transactions on Computer Systems*. Also appears as technical report UCB/CSD 87/345, March 1987.

[9] Ousterhout, J., et al. ''A Trace-Driven Analysis of the UNIX 4.2 BSD File System.'' *Proceedings of the Tenth Symposium on Operating Systems Principles*, December 1985, pp. 15-24.

[10] Popek, G. and Walker, B., eds. *The LOCUS Distributed System Architecture*. MIT Press, Cambridge, MA, 1985.

[11] Sandberg, R., et al. ''Design and Implementation of the Sun Network Filesystem.'' *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.

[12] Theimer, M., Lantz, K., and Cheriton, D. ''Preemptable Remote Execution Facilities for the V-System.'' *Proceedings of the Tenth Symposium on Operating Systems Principles*, December 1985, pp. 2-12.

[13] Welch, B. *The Sprite Remote Procedure Call System*. Technical report UCB/CSD 86/302, June 1986.

[14] Welch, B., and Ousterhout, J. ''Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.'' *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986, pp. 184-189.

[15] Zayas, E. ''Breaking the Process Migration Bottleneck.'' To appear, *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987.