Naming, State Management, and User-Level Extensions in the Sprite Distributed File System

Brent Ballinger Welch

CHAPTER 1

Introduction

This dissertation concerns network computing environments. Advances in network and microprocessor technology have caused a shift from stand-alone timesharing systems to networks of powerful personal computers. Operating systems designed for stand-alone timesharing hosts do not adapt easily to a distributed environment. Resources like disk storage, printers, and tape drives are not concentrated at a single point. Instead, they are scattered around the network under the control of different hosts. New operating system mechanisms are needed to handle this sort of distribution so that users and application programs need not worry about the distributed nature of the underlying system.

This dissertation explores the approach of centering a distributed computing environment around a shared network file system. The file system is chosen as a starting point because it is a heavily used service in stand-alone systems, and the read/write paradigm of the file system is a familiar one that can be applied to many system resources. The file system described in this dissertation provides a distributed name space for system resources, and it provides remote access facilities so all resources are available throughout the network. Resources accessible via the file system include disk storage, other types of peripheral devices, and user-implemented service applications. The resulting system is one where resources are named and accessed via the shared file system, and the underlying distribution of the system among a collection of hosts is not important to users. A ‘‘single system image’’ is provided where any host is equally usable as another, much like a timesharing system where all the terminals provide equivalent access to the central host.

The file system makes a basic distinction between operations on the file system name space and operations on open I/O streams. The name space is implemented by a collection of file servers, while I/O streams may be connected to objects located on any host. Thus, two different servers may be involved in access to any given object, one for naming the object and one for doing I/O on the object. This creates a flexible system where devices and services can be located on any host, yet the file servers’ directory structures still make up the global name space. At the same time, a file server handles both naming and I/O operations for regular files in order to optimize this important common case. The distinction between naming and I/O has been made in other systems by introducing a separate network name service. However, the name services in other systems have been too heavy weight to use for every file. Name services have been used to name hosts, users, and services instead. In contrast, the work here extends the file system name space to provide access to other services. This achieves the same flexibility provided by a separate name service, but it reuses the directory lookup mechanisms already present on the file servers and it optimizes the important case of regular file access.

Another novel feature of the Sprite file system concerns distributed state management. The file system is implemented by a distributed set of computers, and its internal state is distributed among the operating system kernels at the different sites. Much of the state is used to optimize file system accesses by using a main-memory caching system. The servers keep state about how their clients are caching files, and they use this state to guarantee a consistent view of the file system data [Nelson88b]. It is important to maintain this state efficiently so the performance gains of caching are not squandered, yet it is also important to maintain the state robustly so the system behaves well during server failures and network partitions. The dissertation describes how server state is maintained efficiently during normal operations and how server state is updated when processes migrate between hosts. This dissertation also describes a recovery protocol for ‘‘stateful’’ servers that relies on redundant state on the clients. The redundant state is maintained cheaply as main-memory data structures (there is no disk logging), and servers can recover their state after a crash with the help of their clients.

The next section of this introductory chapter gives a little background on the Sprite operating system project. Then the issues addressed by the file system are considered in more detail, along with the more specific contributions I make regarding these issues. The chapter ends with an outline of the other chapters and a summary of the thesis.

1.1. The Sprite Operating System Project

The work presented here was done as part of the Sprite operating system project [Ousterhout88]. The project began in 1984 with the goal of designing and developing an operating system for a network of personal workstations. We felt that new hardware features changed our computing environment enough to warrant exploration of new operating system techniques. The new hardware features include multiprocessors, large main memories, and networks. Of these, my work mainly concerns the network and the problems associated with integrating a network of workstations into a single system.

Sprite began as part of the SPUR multiprocessor project [Hill86]. The SPUR machine is an example of the new technology we felt demanded a new operating system. It is a multiprocessor with a large physical memory, and it is networked together with many other powerful personal workstations. While the SPUR machine was being built by fellow graduate students, our group began development of Sprite on Sun workstations. Today Sprite runs on Sun3 and Sun4 workstations, DECstations, and, of course, multiprocessor SPURs.

Our approach to designing and building Sprite was to begin from scratch. Instead of modifying an existing operating system such as UNIX[1], we decided to start fresh so as to be unfettered by existing design decisions. We were pragmatic, however, and realized that we did not have the man-power to rewrite all the applications needed to build a

[1] UNIX is a registered trademark of A.T.&T.

useful system. We decided to implement the operating system kernel from scratch, but provide enough compatibility with UNIX to be able to easily port UNIX applications. We wanted to provide a system that was like a stand-alone timesharing system in terms of ease of use and ease of information sharing, yet provided the additional power of a network of high-performance personal workstations. Our target was a system that supported a moderate number of people, perhaps an academic department or a research laboratory. Furthermore, our system had to efficiently support diskless workstations, which we favor because they reduce cost, heat, noise, and administrative overhead. Our goal of building a real system has been met. Sprite is in daily use by a growing number of people, and it is used to support continued development of Sprite itself.

There are two features of Sprite that dovetail with its shared file system to provide a high-performance system: a high-performance file caching system, and process migration. The caching system keeps frequently accessed data close to the processes that use them, while the migration system lets processes move to idle workstations. (These features are described briefly below.) Furthermore, the stand-alone, ‘‘timesharing’’ semantics of the file system are maintained so that the additional performance provided by these features does not impact the ease of use of the system.

Sprite uses main-memory caches of file data to optimize file access. The caches improve performance by eliminating disk and network accesses; data is written and read to and from the cache, if possible. Furthermore, a delayed-write policy is used so that data can age in the cache before being written back. In a distributed system this caching strategy creates a consistency problem. Data may be cached at many sites, and the most recent version of a file may not be in the local cache. The algorithm used to maintain cache consistency and the performance benefits of the caching system have been described in Nelson’s thesis [Nelson88b] and [Nelson88a]. The caching system is relevant to this thesis because it creates a state management problem. The state that supports caching has to be managed efficiently so as not to degrade the performance gained by caching, yet it has to be managed robustly in the face of server crashes.

Sprite provides the ability to migrate actively executing processes between hosts of identical CPU architecture. This feature is used to exploit the idle processors that are commonly found in a network of personal workstations; migration is used to off-load jobs onto idle hosts. The process migration system will be described in Douglis’s thesis [Douglis90], while this dissertation describes the effect that process migration has on the file system. The shared file system makes process migration easier because data files, programs, and devices are accessible at a migrating process’s new site. However, the state that supports caching and remote I/O streams has to be updated during migration, and this is a tricky problem in distributed state management. Thus, the combination of data caching and process migration pose new problems in state management that I address in this dissertation.

1.2. Thesis Overview

This dissertation makes original contributions in the areas of distributed naming, remote device access, user-level extensions, process migration, and failure recovery. A software architecture that supports these different features in a well-structured way is also a contribution of this work. Additionally, I present a follow-on study of Sprite's caching system that describes our experiences with it as Sprite has grown into a system that supports real users. These topics are discussed in the following sub-sections.

1.2.1. The Shared File System

The Sprite distributed file system provides a basic framework through which many different kinds of resources (files, devices, and services) are accessed. This is in contrast with other distributed systems where a separate name service is used to locate file servers, device servers, and other service-like applications [Wilkes80] [Birrell82] [Terry85]. In Sprite, the file servers play the role of the name servers in the system. They provide names for devices and services that can be located on any host in the network. The advantage of this approach is that the naming, synchronization, and communication infrastructure required to support remote file access can be reused to provide access to remote devices and remote services. For example, the Sprite file system includes completely general remote waiting and synchronization primitives; the **select** call can be used to wait on any combination of files, devices, and services that are located throughout the network. At the same time, Sprite optimizes access to regular files because only a single server (the file server) is involved; the overhead of invoking a separate name service is eliminated.

The internal file system architecture makes a fundamental distinction between naming operations and I/O operations. Naming operations are performed in a uniform way for all file system objects by the file servers, and I/O operations are object-specific and they may be implemented by any host. Thus, there are three roles that a host can play in the architecture, the file server that does naming operations, the *I/O server* that implements object-specific operations, and the client that is using the object. The architecture supports this by defining two main internal interfaces, one for naming operations and one for I/O operations. Different implementations of these interfaces cleanly support the different cases that the system has to address. A diskless workstation, for example, names local devices via a remote file server, but it implements I/O operations on the device itself. While other file system architectures define generic internal interfaces [Kleiman86] [Rodriguez86], they are oriented towards file access and have limitations that preclude the general remote device and remote service access provide by Sprite.

I developed a distributed name resolution protocol that unifies the directory structures of a collection of file servers into a uniformly shared, global name space. Clients keep a *prefix table* that is a mapping from file name prefixes to their servers. The prefix tables are caches that are updated with broadcast protocol; new entries are added as a client accesses new areas of the file system, and out-of-date entries are refreshed automatically if the system configuration changes. This naming system is more dynamic than the static configurations used in most UNIX-based systems, and therefore it is easier to manage. The shared name space and its adaptive nature makes it easy to add servers, move files between servers, and add new disk storage. The naming system is optimized towards the local area network environment. It uses a light-weight caching- and broadcast-based system instead of the heavier-weight replicated databases used for larger scale name services [Birrell82][Popek85] [Kazar89].

1.2.2. Distributed State Management

The Sprite file system is a tightly-coupled distributed system of servers and clients. The servers are ‘‘stateful’’; they keep state about how their resources are being used by other hosts. The motivation for the state is to preserve the semantics of a stand-alone system where the file system is implemented in a single operating system kernel. For example, the Sprite caching system provides high-performance access to remote data, yet it is transparent to application programs because the file servers can guarantee a consistent view of the file system by keeping state about how remote clients are caching data. This dissertation makes two contributions regarding the management of this state. The first concerns updating the state during process migration, and the second concerns maintaining the state across server failures.

Process migration causes two problems that have to be solved. The first problem is to simply update the file system's state to reflect the migration of a process. This problem is complicated by the concurrency present in the system. Different processes can share I/O streams and perform different operations on them concurrently. I present a deadlock-free algorithm to update the file system's distributed state during migration. Process migration also causes a problem with shared I/O streams. Processes that share an I/O stream also share the current access position of the stream which is maintained by the kernel. When migration causes processes on different hosts to share a stream then it is no longer possible to maintain the current access position in a single shared kernel data structure. I describe a system based on *shadow stream descriptors* kept on the file servers that is simpler than the token-passing schemes used in other systems.

The other contribution I make regarding distributed state management concerns maintaining the state during server failures and network partitions. I developed a new state recovery protocol based on the principal of keeping redundant state on the clients and servers. The state is kept in main memory data structures so it is cheap to maintain during normal operation. This is in contrast to other recovery systems that are based on logging state to stable storage. In Sprite, the redundant state allows the servers to rebuild their state with the help of their clients. This approach also makes the system robust to diabolical failure modes that arise during network partitions. The recovery protocol is idempotent so it can be invoked by clients any time they suspect that the system's state has become inconsistent due to a communication failure. Common cases like server reboots are handled invisibly to applications, and diabolical failure modes are detected and conflicting recovery actions are prevented. The recovery system also gives users the option of aborting jobs that access dead servers instead of waiting until the server is rebooted.

Finally, I present a follow-on study that examines the behavior of the caching system on our live Sprite network. The caching system is the main cause of the state management problem, so its performance benefits ought to be significant in order to justify the additional complexity of the system. Data presented in Chapter 8 indicates that about 50% of file data is never written back to the Sprite file servers because it is overwritten or deleted before the 30-second aging period expires. Read hit ratios on the client caches are from 30% to 85%, averaging 75% across all clients. However, paging traffic on clients with small memories can dominate traffic from file cache misses. The variable-sized caches occupy over 50% of the file server's main memory, and from 15% to 35% of the clients main memory. However, the read hit ratios on the servers are reduced because the clients' caches are so effective. The server has to have an order of magnitude more memory than the clients before its cache becomes as effective as the clients'.

1.2.3. Pseudo-Devices and Pseudo-File-Systems

The third main area that is addressed in this dissertation concerns the extensibility of the system. Sprite provides mechanisms that allow system services to be implemented by non-privileged, or ''user-level'', server processes. While performance-critical services are implemented within the Sprite kernel, user-level services provide a way to easily add functionality to Sprite. The user-level services are implemented as application programs using the standard development and debugging environment. The motivation for user-level implementation is to avoid uncontrolled growth of the operating system kernel as new features are added to the system over time. Keeping the kernel small means it is more likely to remain stable and reliable. While other systems such as Mach [Accetta86] and the V system [Cheriton84] support user-level services, the approach taken in Sprite is novel because the user-level services are integrated into the distributed file system. The user-level services in Sprite benefit from kernel-resident features such as the distributed name space and remote access facilities, which are present to support remote file and device access.

The server processes appear in the file system as *pseudo-devices*[Welch88]. Pseudo-devices have names in the file system, but I/O operations on a pseudo-device are forwarded by the kernel up to a user-level server process. Access to remote pseudo-device servers is handled by the kernel mechanisms that support remote file and device access. The pseudo-device mechanism replaces the **socket** operations provided in UNIX, plus it moves the protocol processing out of the kernel. Pseudo-devices are used to implement an Internet Protocol server and an X window server for Sprite. A user-level server process can also implement a *pseudo-file-system* by handling naming as well as I/O operations. Sprite uses a pseudo-file-system to provide access to NFS[2] file servers. The NFS directory structures are transparently integrated into the distributed file system via the pseudo-file-system mechanism. More interesting applications for pseudo-file-systems are possible. A version control system can be implemented as a pseudo-file-system that automatically manages different versions of files. Or, an archive service can be implemented as a pseudo-file-system that presents a directory structure that reflects the time a file was archived. Pseudo-devices and pseudo-file-systems retain the standard file system interface, yet they allow extensions to the system to be implemented outside

[2] NFS is a trademark of Sun Microsystems.

the kernel.

An important lesson to learn from user-level services is that there is a significant performance penalty in comparison with kernel-based services. Performance studies of the Sprite mechanisms indicate that the penalty can range from 25% to 100% or more. The reduced performance is due to unavoidable costs associated with switching between user processes as opposed to merely trapping into the kernel. Consequently, there is a fundamental trade-off between having a higher-performance service (and a bigger kernel) and having a service that is easier to develop and debug. While other systems take one approach or the other (kernel-based services or user-level services), in Sprite both options are available. The heavily used services, such as regular file access, are kernel-resident in Sprite, while user-level services are used for less performance-critical services such as remote login, mail transfer, and newly developed services. Moving a pseudo-device implementation into the kernel is not automated, but it is quite feasible because it shares the file system's main internal interface.

1.3. Chapter Map

The remaining chapters are organized as follows. Chapter 2 provides background information and describes related work in file systems and distributed systems. Chapter 3 describes the basic architecture of the file system, focusing on the distinction between naming and I/O and on the remote synchronization primatives. The remaining chapters consider individual problems in more detail.

Chapter 4 describes the distributed naming mechanism, which is based on *prefix tables* and *remote links* [Welch86b]. The key properties of the system are that it supports diskless workstations, it merges the directory hierarchies of Sprite file servers and pseudo-file-system servers into a seamless hierarchy, and it dynamically adapts to changes in the file system configuration.

Chapter 5 describes the impact that process migration has on the file system. The state maintained by the file servers to support caching has to be updated during migration. This is complicated by concurrency that results from shared I/O streams. The semantics of shared I/O streams also have to be preserved when the processes sharing the stream are on different hosts. This is done via shadow stream descriptors that are kept on the I/O server.

Chapter 6 describes the way the system's state is organized to be fault-tolerant. Some of the servers' state is duplicated on the clients in order to facilitate crash recovery via an idempotent *re-open* protocol. This chapter also describes the low-level crash detection system and shows that it adds little overhead to the system.

Chapter 7 describes the pseudo-device and pseudo-file-system mechanisms. A ‘‘request-response’’ protocol is used to forward operations from the kernel up to the user-level server. The protocol provides read-ahead buffering and asynchronous writing to reduce the number of interactions with the server. Even without buffering the pseudo-device connection is about as cheap as pipes, and much faster than a UNIX TCP connection in the remote case.

Chapter 8 reviews the caching system and provides measurements of its performance on our network. The performance is of interest in itself, and it also motivates the distributed state management problem described in the previous chapters.

The final chapter provides some conclusions and a review of the thesis. There are 4 appendices. Appendix A is a short retrospective of the Sprite project. Appendix B presents some additional measurements that are too detailed to be included in the other chapters. Appendix C describes the Sprite RPC protocol. Appendix D describes the RPC interface to Sprite.

1.4. Conclusion

There are three main themes to the thesis. The first theme is transparency with regard to the network. A distributed system is only a hindrance unless the operating system takes measures to manage the distribution on behalf of its users and applications. This idea of network transparency has been promoted in other systems, most notably LOCUS [Popek85]. Like LOCUS, Sprite file system operations apply uniformly whether they are implemented locally or by a remote server. LOCUS is oriented towards a system of a few timesharing hosts. It focuses of file replication and provides limited remote device access. Sprite is oriented towards a system of diskless workstations and their servers. Sprite focuses on file caching for high-performance, and it provides very general remote access that naturally extends to devices and pseudo-devices.

The second theme concerns maintaining the state of the distributed system in the face of change. The system has to keep its internal state, which is distributed among the operating system kernels on all the hosts, consistent in the face of host failures and during the migration of processes from one host to another. Previous systems have relied on stable disk storage, which is a potential performance bottleneck, especially as CPU power continues to accelerate past disk performance. Other systems propose replicated servers. TANDEM [Bartlett81] is a good example of a real system that uses replication. Replication of servers is expensive, however, both in performance and equipment. The cost of replication is only justifiable for certain transaction processing systems. Instead, Sprite uses stable storage for a few pieces of critical state, and then it duplicates performance-critical state in the main-memory of different hosts. The result is a high-performance distributed file system with good failure semantics.

The final theme of the dissertation concerns extensibility. It is not practical or desirable to implement all system services inside the operating system kernel. Sprite includes mechanisms that allow user-level processes to implement file-like objects (i.e., pseudo-devices) and directory hierarchies (i.e., pseudo-file-systems). These provide a convenient framework for implementing a variety of services outside the kernel. However, unlike many modern operating systems, Sprite does not move all high-level services out of the kernel. The most heavily used services, in particular file and device access, are kept in the kernel to optimize performance. Furthermore, because the internal I/O and naming interfaces are exported to the user-level servers, their functionality can be moved into the kernel (or out of it) depending on performance requirements.

The common thread to all of these themes is the file system interface. All objects, whether they are local or remote, system or user-implemented, are named and accessed in the same way. The system provides basic functionality common to all (e.g., crash recovery, process migration, blocking I/O). Thus, the system is composed of several well-integrated features that combine to provide a high-performance, robust, distributed computing system.

CHAPTER 2

Background and Related Work

2.1. Introduction

This chapter presents background information on file systems, distributed systems, and the Sprite operating system. While the dissertation concerns issues that stem from the network environment, it is important to have a basic understanding of stand-alone file systems. This chapter describes file system features such as a hierarchical directory system, device-independent I/O, and data caching. Basic concepts of distributed systems are also presented, including communication protocols, name services, and the client-server model. Sprite has similar goals as other distributed systems such as LOCUS[Popek85] and the V-system [Cheriton84], and the related work section describes the approaches taken by these and other systems. Finally, some overall ideas about the scale of the distributed system, location-transparency, and the issue of stateful servers are discussed.

2.2. Terminology

Here are definitions for some terms I use throughout the dissertation:

Host — A host is a computer on the network.

Kernel — The kernel is the memory-resident code of the operating system that executes in a privileged state. A distributed operating system like Sprite is composed of the kernels on each host.

Application — An application is a program that a user runs in order to perform some task. Examples include text editors, compilers, and simulators.

Process — A process is an executing program. It can be a user-level process that is part of an application. An application can be composed of several processes. There are also kernel processes in Sprite; the kernel has several processes used to service network requests and do background processing.

System Call — A system call is a special procedure call that a user-level process makes to invoke operating system functions. During a system call the process continues execution inside the kernel in a privileged state. There is a well-defined set of system calls used to open files, read the clock, create

	processes, etc.[3]
Object	Object is a general term for things accessed via the file system interface. It can refer to a regular file, directory, device, or pseudo-device.
File	A file is the basic object maintained by the file system. A file has some attributes like its type and ownership. A file may have disk storage associated with it.
Device	A device is an attached peripheral such as a tape drive, printer, display, or network interface.
Device File	A device file is a special file that represents a device. The device file has no associated data storage, but its attributes define the type and location of the device. The name of the device file is used to reference the device.
Pseudo-Device	A pseudo-device is an object implemented by a user-level server process. Operations on a pseudo-device are transparently forwarded to the server process by the kernel. Pseudo-devices are represented by special files in the file system so they have names and attributes.
Directory	A directory is a special file used by a file server to record the correspondence between object names and disk files. The files referenced by a directory may be ordinary files, other directories, or special files that represent devices or pseudo-devices.
Client	In general, a client is an entity that is receiving service from some other entity. Most often I use this term to refer to the operating system kernel when it needs service from a kernel on another host. Occasionally, especially in Chapter 7 on user-level services, ‘‘client’’ will refer to an application program that is requesting service from another application.
Client Kernel	A Sprite kernel that is requesting service from a remote kernel, i.e. to open a file for an application or to write out a dirty page from the virtual memory system.
Server	A server is an entity that provides some function to its clients. Again, this term most often refers to the operating system kernel on a server machine. In Chapter 7, ‘‘server’’ is used to refer to a user-level service application.
File Server	A file server is a host with disks that store a part of the distributed file system. In addition to providing access to files, the Sprite file servers implement the directory structure used for naming.
I/O Server	An I/O server is the kernel that controls access to an object. In general the I/O server for an object and the file server that names an object may

[3] The Sprite system call interface is very similar to UNIX 4.3BSD. For historical reasons it is not binary compatible, but a simple compatibility library is linked with standard UNIX programs so that they can run on Sprite.

be different.

I/O Stream — A stream is an I/O connection between a process and an object in the file system. The term ‘‘stream’’ refers to data transfer between the object and the process. A process may also manipulate an object via an I/O stream in other ways such as truncating the length of a file or rewinding a tape device.

2.3. The UNIX File System

The UNIX file system is important because it has several features that make it easy to use and share information[Ritchie74], and it is widely used today. By adopting the UNIX file system interface Sprite benefits from several good design features and a large base of existing applications. While Sprite retains the UNIX interface, the implementation of the Sprite kernel was done completely from scratch, and its internals differ markedly from other UNIX implementations. This section reviews the major features of the UNIX file system that relate to the rest of the dissertation.

2.3.1. A Hierarchical Naming System

The UNIX file system is organized as a hierarchical tree of directories and files. The tree begins at a distinguished directory called the root. A hierarchical naming system is created by allowing directory entries to name both files and directories. This recursive definition of the name space originated in Multics [Feirtag71] and is used in UNIX and most modern operating systems.

Objects have a *pathname* that is a sequence of *components* separated by slash characters (e.g., ‘‘/a/b/c’’). The root directory is named ‘‘/’’. *Name resolution* is the process of examining the directory structure in order to follow a pathname. It is an iterative procedure where the current directory is scanned for the next component of the pathname. If the component is found and there are more components left to process, then the current directory is advanced and the lookup continues. The lookup terminates successfully when the last component is found. For example, the pathname ‘‘/a/b/c’’ defines a path through the hierarchy that starts at the root directory, ‘‘/’’, and proceeds to directory ‘‘a’’, and on to directory ‘‘b’’, and ends at object ‘‘c’’. ‘‘/a/b/c’’ can be the pathname of a file, directory, device, pseudo-device, or a symbolic link to another pathname. (Symbolic links are explained below.)

As a convenience to users, UNIX provides a *current working directory* and *relative* pathnames. Relative pathnames implicitly begin at the current working directory, while *absolute* pathnames begin at the root directory. Relative pathnames are distinguished because they do not begin with ‘‘/’’, the name of the root directory. For example, if the current directory is ‘‘/a/b’’, then the object ‘‘/a/b/c’’ can be referred to by the relative pathname ‘‘c’’. The current working directory can be changed by a process in order to make it easier to name commonly used files. Relative names are shorter, which means they are both easier to type by the user and more efficient to process by the operating system.

The combination of relative pathnames and the specification of the parent directory means that a pathname can begin at any point in the hierarchy and travel to any other point. A directory's parent may be referred to with the name ''..''. For example, if the current working directory is ''/a/b/c'', then ''..'' refers to ''/a/b''. Sibling directories are easily referenced with relative names (e.g., ''../sibling''). Thus, relative pathnames can be used within a set of related directories so that the absolute location of the directories isn't important, only their relative location.

The hierarchical structure of the name space can become more arbitrarily connected by using *symbolic links*. A symbolic link is a special file that is a reference by name to another point in the hierarchy. There are no restrictions on the target of a symbolic link, so the plain hierarchy can be converted to a more general directed graph. Symbolic links are implemented as files that contain pathnames. Traversing a link is implemented by replacing the name of the link with its contents and continuing the normal lookup algorithm. For example, if ''/a/d'' is a symbolic link to ''b/c'' then the pathname ''/a/d/f'' gets expanded to ''/a/b/c/f''. Symbolic links can also cause jumps back to the root. If ''/x/y/z'' is a symbolic link to ''/a/b'', then the pathname ''/x/y/z/c'' gets expanded to ''/a/b/c''. These links are shown in Figure 2-1.

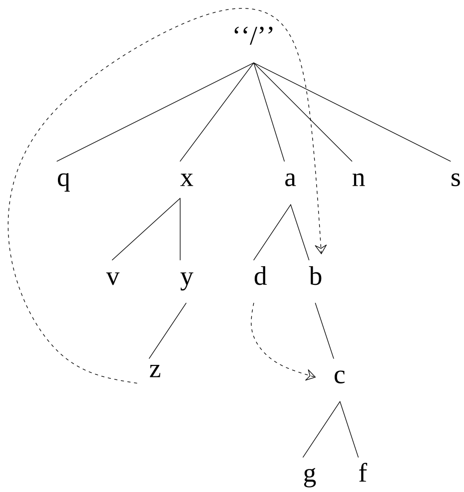


Figure 2-1. Symbolic links in a hierarchical name space. The hierarchical structure of the name space can become arbitrarily connected by using symbolic links, which are depicted as dotted lines in this figure. ''/x/y/z'' is a link to ''/a/b''. ''/a/d'' is a link to ''b/c''.

2.3.2. Objects and their Attributes

The file system is used to access different types of objects. To distinguish among the different object types, the file system keeps a set of attributes for each one. One attribute is its type, (e.g. file, directory, symbolic link, or device). There are also ownership and permission attributes that are used for access control. All objects have these attributes and are subject to the same access controls. Applications can query the attributes of an object, and the owner of an object can change some attributes (like the access permissions). Other attributes, like the size and modify time, are updated by the system as the object is accessed.

It is sometimes important to distinguish between the name of an object and the object itself. Special files are used as place-holders for non-file objects; their entry in a directory gives the object a name. The special files also serve as convenient repositories for attributes of these other kinds of objects. A device file, for example, records the type and unit number of the device, and, in Sprite, it records the host to which the device is attached. These details are ordinarily hidden by the operating system, but in Chapter 3, which describes the internal architecture of the Sprite file system, it is important to understand the difference between the representation of an object's name and the object itself.

2.3.3. I/O Streams

An *I/O stream* represents a connection between a process and an object in the file system.[4] A stream is either created by naming a file system object with the **open** operation, or, as described below, a stream may be inherited. Each I/O stream has a current access position that is maintained by the kernel on behalf of the processes using the stream. Each **read** and **write** operation updates the stream's current access position so that successive I/O operations apply to successive ranges of the object. This supports the common method of file system accesses in which the whole object is read or written, and it eliminates the need for an explicit location (or ‘‘offset’’) parameter to the **read** and **write** system calls.

2.3.3.1. Standard I/O Operations

The following UNIX operating system calls apply uniformly to I/O streams.

open This operation follows a pathname and sets up an I/O stream to the underlying object for use in the remaining operations. Access permissions are checked on the directories leading to the object and on the object itself.

---

[4] Sprite implements the UNIX concept of an I/O stream, which includes stream inheritance and stream sharing. The standard UNIX terminology is ‘‘open file descriptor’’, however, instead of ‘‘I/O stream’’.

read This operation transfers a number of bytes from the object into a buffer in the application's address space. The operating system maintains a current access position for a stream that indicates what byte is the next one to read. Reading N bytes advances the access position by N so that successive reads on a stream advance sequentially through the object's data.

write This operation transfers a number of bytes to the object from a buffer in the application's address space. The access position is advanced in the same way that the read operation advances it.

select This operation blocks the process until any one of several streams are ready for I/O, or until a timeout period has expired. **Select** is useful for servers that have several streams to various clients, or to window system clients that have a different stream for each window they use.

ioctl This operation provides a general hook to get at object-specific functionality. The parameters to **ioctl** include a command specifier, an input buffer, and an output buffer.[5] Example operations include rewinding tape drives and setting terminal erase characters. Sprite also uses **ioctl** to implement a number of miscellaneous UNIX system calls such as **ftrunc**, which truncates a file, and **fsync**, which forces a cached file to disk.

fstat This returns the attributes of the object to which the stream is connected. This includes the object's type (file, device, pipe), and type-specific attributes like the size of a file.

close This operation removes a reference to an I/O stream, and after the last reference is gone the stream is destroyed. Additional references arise during process creation because streams are shared between parent and child.

2.3.3.2. Stream Inheritance

Each UNIX process begins execution with access to three standard I/O streams: standard input, standard output, and standard error output. Instead of having the operating system kernel set up standard streams for a new process, UNIX uses a more general mechanism of stream inheritance. When a process is created it inherits its I/O streams from its parent process. The parent is free to set up any configuration of I/O streams for the child. UNIX command interpreters, which are the parent processes of user commands, ensure that the standard streams are set up.

Inheritance is useful with processes that are independent of what their input and output I/O streams are associated with. Command interpreters provide the capability to

---

[5] **ioctl** in Sprite is a super-set of UNIX **ioctl**. UNIX **ioctl** only uses a single buffer and its size is implicit. Sprite **ioctl** has two buffers, one for input and one for output, and their sizes are explicit. This extension makes it easier to pass the **ioctl** operation around in a general way (e.g to remote hosts or to pseudo-device servers).

redirect the standard streams to files and devices so that users can specify on the command line where input and output should go, and the program does not have to worry about it. For example, program output might go to a file, to a peripheral device like a printer, or to another process which transports the data across the country via a specialized internetwork protocol. This notion of *object-independent I/O* originated in Multics (as ‘‘device-independent I/O’’ [Feirtag71]) and was adopted by UNIX.

A subtle, but important, side effect of stream inheritance is the fact that an inherited stream is actually shared between the parent and the child process. Specifically, the stream's current access position is shared; I/O operations by any process sharing the stream advance the stream's current access position. This property is most often used with command files (‘‘scripts’’ in UNIX). Each command is a child process of a common parent. The parent process shares the input and output streams with each child in succession, and each child process advances the stream offsets as it does I/O. Output of one process will not overwrite output of a previous process, and input processed by one process will not be re-read by the next process. (Read-ahead buffering by library I/O routines, however, can cause one process to steal input from the next one.) Preserving these semantics of shared I/O streams can be tricky in a distributed environment, and the way this is done in Sprite is described in Chapter 5.

2.3.3.3. Pipes

A *pipe* is a one-way communication channel that connects the writing stream of one process to the reading stream of another process. Ordinary UNIX pipes have no name in the file system.[6] Instead, two I/O streams, one for reading from the pipe and one for writing to it, are created with the **pipe** system call. Parent processes can use stream inheritance to pass these streams to child processes. UNIX command interpreters make it easy to compose a set of processes into a pipeline, with the output of the first hooked to the input of the second, and so on. There are a number of standard filter programs in UNIX that can be composed into useful pipelines. They include programs that sort, edit, select lines based on pattern matching, compress and decompress.

2.3.4. The Buffer Cache

In order to reduce the latency associated with disk accesses the file system maintains a block-oriented cache of recently accessed file data. The cache is checked before each disk read, and if the block is in the cache then a disk access is avoided. When data is written it is placed in the cache and written back in the background so applications don't have to wait for the disk operation. Data is allowed to age for 30 seconds before being written to disk. Studies by Ousterhout [Ousterhout85] and Thompson [Thompson87] show that many temporary files get created, read, and deleted within this time

[6] UNIX System V has *named pipes*, which do have names in the file system.

period so their data never gets written to disk.

2.3.5. UNIX File System Summary

To summarize, the important features of the UNIX file system include its hierarchical name space, object-independent I/O with its open-close-read-write interface, and the main-memory buffer cache to optimize performance. Sprite retains these features in its distributed environment, although it often employs different mechanisms and techniques than a stand-alone UNIX system.

2.4. Distributed System Background

The two fundamental problems introduced by a distributed system are host-to-host communication, which is addressed by network communication protocols, and resource location, which is addressed by name servers that map resource names to network locations and other resource attributes. The client-server model is commonly used to structure a distributed system. In this case servers are identified by the name service, and a standard communication protocol is used between clients and servers. These topics are discussed in more detail below.

2.4.1. Network Communication

Communication protocols provide a standard way to communicate between hosts connected by a network. Protocols address such issues as routing a message through the network and reliable transmission of messages over an unreliable network. A range of protocols exist that vary in their reliability and efficiency. Cheap but unreliable *datagram* protocols [IP81] are used to build up more reliable (and more expensive) protocols such as *virtual circuit* and *request-response* protocols. A virtual circuit provides a reliable data channel between two hosts that is useful for bulk data transfer and long-lived connections [TCP81]. Virtual circuit protocols are usually optimized towards increasing *throughput*, maximizing the rate at which large amounts of data can be transferred. The request-response protocols [Birrell84] are oriented towards service applications where clients make requests of servers and the servers return some response. These protocols are usually optimized towards reducing *latency*, minimizing the time a client has to wait for a response from the server.

2.4.2. The Client-Server Model and RPC

A standard way to structure a distributed system is to have servers that control resources and clients that need to access these resources. A standard communication protocol enables a client to access any server in the system. The least structured and most flexible communication protocol is a datagram protocol in which a message is sent from Client A to Server B (or vice versa). In practice, however, it turns out that a more stylized request-response protocol is used between clients and servers. The client issues a request message to the server and blocks itself awaiting a response message from the server. Upon receipt of a request message the server performs some function and returns the results to the client in a response message. The reason that request-response communication is preferred is that it resembles what happens in a programming language during a procedure call. The caller suspends itself as it invokes a procedure, and it continues after the procedure completes. By introducing *stub procedures* that handle packaging of parameters into messages and the use of the request-response protocol, the client can use a regular procedure call interface to access the server. This technique is called Remote Procedure Call (RPC) [Birrell84].

There are two main benefits from the use of RPC. The first is that by retaining the procedural interface it is efficient to access local resources. In the local case, the client executes the service procedure directly and there is no need to switch to a server process. In contrast, an explicit message-based interface between the client and the server adds extra overhead from messages and process switching even in the local case. The second benefit is that RPC encourages a clean system design because the effects of message passing are confined to the stub procedures and the request-response protocol. The client and the server are basically unaware of the complexities associated with the message protocol. The internal Sprite file system architecture, which is described in detail in the next chapter, relies on these aspects of RPC. Each Sprite kernel communicates with other kernels via a kernel-to-kernel RPC protocol when it needs access to remote resources [Welch86a].

2.4.3. Resource Location

Most distributed systems have a *name service* as one of their fundamental components [Terry85][Schroeder84][Oppen83][Needham79]. The role of the name service is to provide a registry for all services in a distributed system. The process controlling a resource (e.g., a file system or printing device) registers itself with the name service, and other processes can query the name service in order to determine the correct process with which to communicate. Thus, using a resource in a distributed system is divided into two steps: 1) an initial *binding* step in which the resource is located and 2) the subsequent use of the resource by exchanging messages with the server that controls the resource.

One disadvantage of using a global name service is that it adds overhead; the name server must be contacted first to locate a particular service before the service can be invoked directly. Because of this, name servers have been used to locate things like hosts, people, and other services, but name servers are not typically used on the fine grain needed for a file system, i.e. for each file in the system. This results in a two-level naming system (i.e. service-name:object-name) and perhaps also a duplication of effort between each service, which has to interpret its object names, and the name service, which has to interpret service names.

In Sprite, the file system name space is used as the name service [Welch86b]. The binding between client and server is done as part of the **open** system call, and the **read**, **write**, and **ioctl** calls are used to access the service (e.g., files, devices, and pseudo-devices). This will be explained in detail in Chapter 3. This approach optimizes the common case of file access because there is only a single server involved; the overhead of accessing a separate name service is eliminated.

2.5. Related Work

This section describes related work in distributed systems, including early remote file access systems, message-passing systems, and distributed file systems.

2.5.1. Early Distributed Systems

An early approach to providing a distributed system was to glue together existing stand-alone systems with a simple name service. This approach was taken in systems such as UNIX United [Lobelle85] [Brownbridge82] and the Cocanet system [Rowe82] by modifying file name syntax to explicitly identify the server (e.g., ‘‘hostA:/a/b/c’’). The runtime library was modified to check for file names with the special syntax, and a special library routine was invoked to forward the operation to a remote host. This approach is clean and simple if the operating system interface is also simple because there may be only a few points in the library that need modification. The disadvantage of modifying name syntax is that the distribution of the system is evident in the name space. Full names are ‘‘server:name,’’ so it is not possible to move files among hosts without changing their full names.

2.5.2. Message-Based Systems

Several research operating systems have been developed that focus on communication protocols as a basic system component. In these *message-based* systems, all processes interact by exchanging messages. The operating system provides the capability to reliably send messages to processes executing on any host in the network [Powell77] [Bartlett81] [Fitzgerald85]. This represents a distinct change in system model from the traditional operating systems that provide each process with an isolated *virtual machine* upon which to execute[Goldberg74]. Instead, a process in a message-based system interacts asynchronously with other processes and the operating system itself by sending and receiving messages.

2.5.2.1. The V System

The V system[Cheriton84] has focused on a simple, high-performance message passing facility as one of its fundamental building blocks. The V kernel only implements address spaces, processes, and the interprocess communication protocol. All higher-level system services are implemented outside the kernel in separate processes. The argument to support this approach is that because the operating system kernel is simpler it can be more efficient, and because most system services are outside the kernel it is easier to experiment with new implementations of these services. An important common denominator among the V servers is a naming protocol so that all system services can be named in a similar way [Cheriton89]. The naming protocol distinguishes among different servers with name *prefixes* such as ‘‘print’’ and ‘‘fileserverA’’. A complete name is a prefix and a name (e.g., ‘‘fileserverA]a/b/c’’). The ‘]’ character distinguishes the prefix, and a global name service is consulted to locate the server process corresponding to this prefix. The remainder of the name is evaluated by the server itself. The V system also has a Uniform I/O protocol [Cheriton87] for those services that provide access to files and peripheral devices. This protocol defines standard message formats that are used to **read** and **write** data from and to the service.

2.5.2.2. Mach

Mach [Accetta86] is another message-passing system that is designed to support a set of user-level server processes. The focus of Mach, however, has been on the use of virtual memory techniques (i.e., memory mapping) to optimize interprocess communication. Processes can share memory regions directly, or they can exchange messages as is done in the V system. Virtual memory mapping techniques are used to optimize message passing between processes on the same host. This approach is oriented towards multiprocessors with a globally shared memory, and it does not work as well in a distributed system. Mach invokes a user-level server process to implement its network communication protocols, and user-level server adds extra overhead [Clark85]. In contrast, Sprite implements its RPC protocol within the kernel so the remote case is still relatively efficient.

2.5.2.3. Amoeba

The Amoeba distributed system [Renesse89] is a message-based system that is distinguished by its use of *capabilities* for access control and message addresses. A capability is an encrypted token that represents some service or process [Fabry74], and the operating system provides the ability to send a message to the process that issued the capability. The capability also encodes access permissions so that the service can control use of its resources. A name server provides a directory service that maps names to capabilities.

2.5.3. Remote File Services

A frequently used service provided by the operating system is the file system. The file system provides long term storage for program images and data files, so virtually all other services and applications depend on it. Because of this, most distributed systems include some form of remote file access capability. Sprite takes this approach to the extreme by making its distributed file system the foundation for the rest of the system. A number of other distributed file systems are reviewed below.

2.5.3.1. WFS

WFS was a simple remote file service that provided page-level access to files stored on a file server[Swinehart79]. Files were created and deleted using a unique identifier (UID), and **read** and **write** operations applied to whole pages of a file. All other file-related operations (e.g., high-level naming, locking, etc.) were provided by other services. An important characteristic of WFS is that it was a ‘‘stateless’’ service; all long term state of the WFS file service was kept on disk so that the server could crash without losing state information. A stateless design simplified the implementation of WFS. Each **write** request was blocked until the data was safely on the disk, and the server could then ‘‘forget’’ about the **write**. WFS is an example of a very low-level service; it simply exports a disk to other hosts on the network.

2.5.3.2. IFS

The ‘‘Interim’’ File Service (IFS) was developed as a follow-on to WFS. The IFS servers were designed to be shared repositories for files in the CEDAR environment [Swinehart86]. Each CEDAR workstation, however, was a stand-alone system with its own own private file system. The private file system was used as a cache of files kept on the IFS servers, and shared files had to be copied to and from the IFS servers as needed. Tools were developed to make this style of sharing less tedious and error prone [Schroeder85], but users still had to be conscious of where files were stored and whether or not they were safely backed up to an IFS server.

2.5.3.3. NFS

NFS is a remote file service developed for the UNIX environment [Sandberg85]. Like WFS, NFS is based on the stateless server model. Consequently, an NFS server doesn’t know about open I/O streams. There is no **open** or **close** operation in the NFS protocol; an NFS server just responds to **read** and **write** requests that identify the file and the user that is making the request. The reason that the stateless server model was adopted by NFS was to simplify error recovery. A stateless server can be restarted at any time without breaking its clients. Clients retry operations indefinitely, even if it takes the server minutes or hours to respond. The stateless nature of the server means it can begin servicing **read** and **write** requests immediately after it restarts. There is no special recovery protocol required to bring an NFS server back on-line.

However, a drawback of the stateless nature of NFS is demonstrated by its caching system. Clients keep a main-memory cache of file data to eliminate some network accesses. Because the NFS server is stateless, however, the caching scheme cannot provide *consistency* while cached files are being updated; it is possible for a client to get an inconsistent view of the file system. The stateless server doesn’t remember what clients are caching files so it is the clients responsibility to keep their caches up-to-date. They achieve this by periodically checking with their server, which leaves windows of time during which file modifications may not be visible at a client. Thus, the stateless nature of NFS prevents it from maintaining the semantics of a stand-alone UNIX file system when it uses caching to improve performance.

2.5.3.4. RFS

RFS is another UNIX-based remote file service[Rifkin86]. It differs from NFS in that the servers do keep state about how their files are being used by clients. Server state is used to support a more sophisticated caching scheme that guarantees consistency. The servers make *callbacks* to clients that indicate that a file is being modified. This eliminates the case where a client fetches stale data from its cache. One problem with RFS, however, is that there is no mechanism to recover the server's state after it crashes. This means that I/O streams to files on a server that crashes are forcibly closed [Atlas86]. In contrast, a stateless NFS server can crash and reboot and I/O operations are only delayed, not aborted.

2.5.3.5. AFS

The Andrew File System (AFS) was developed to support a large scale system, up to several thousand nodes [Satyanarayanan85]. It uses a caching system that keeps files on local disks, which is similar to IFS, instead of a system that caches files in main memory, which is done in NFS and RFS. Unlike IFS, however, AFS automatically manages the file cache so it is transparent to the user. Also, consistency is guaranteed, except during concurrent updates by different clients, by the use of callbacks to the clients. AFS puts a time limit on each callback in order to reduce the state maintained by the server. A server only promises to make a callback within the time limit, and it can discard information about the callback after the time limit expires.[7] One drawback of the caching system used in AFS is that is assumes local disks; diskless clients are not supported.

2.5.3.6. LOCUS

LOCUS[Popek85] is a UNIX-based distributed system is similar to Sprite in many respects. Perhaps the most important goal these systems share is to provide a *transparent* distributed file system, one in which the location of files is not evident or important to users. However, LOCUS was designed to allow sharing among a small number of timesharing hosts, while Sprite was designed to support a larger number of diskless clients and file servers. Different assumptions are made in the design of the two systems.

The most significant assumption is that LOCUS depends on file replication [Walker83a], while Sprite uses file caching. In LOCUS, files are replicated in the file systems of different hosts to increase their availability and improve performance. Certain critical files are replicated in every host's file system so that hosts can remain autonomous. Replication requires a more complex **write** protocol so that all available copies of a file get updated. The primary advantage of replication over caching is that it

---

[7] A callback with a time limit is sometimes referred to as a *lease* [Gray89]. The server leases a file to a client for a given period of time.

improves the availability of the system. Local copies of a file can be used even if a network partition has isolated a host from the rest of the system. However, autonomous operation gives rise to possible conflicts between updates to different copies of a file [Parker83]. Replication requires a recovery protocol so that after a host crash or a network partition the system can ensure that all file system replicas are consistent. The LOCUS recovery protocol requires communication among every host-pair for an overall cost proportional to the square of the number of hosts. Thus, replication is attractive because it can improve availability, but it comes at the cost of a more complex **write** protocol and and expensive recovery protocol.

In contrast, the caching approach taken in Sprite is much lighter-weight. Files are cached in main-memory instead of duplicated on disks so they are efficient to access. **Write** operations are simpler because they do not have to be propagated to all cached copies of a file. Instead, out-of-date copies are invalidated the next time they are used. As described in [Nelson88a], this invalidation requires at most one remote operation. Recovery is also simpler in Sprite. Its cost is only proportional to the number of clients caching a file as opposed to the number-of-hosts-squared cost incurred in LOCUS. The caching approach taken in Sprite simplifies things, and it provides high-performance.

2.5.3.7. Other Systems

There are a number of other remote file systems reported in the literature. Apollo DOMAIN is an early distributed system that is based on the memory-mapped file model [Leach83][Leach82]. There are many UNIX-based systems similar to NFS or RFS [Hughes86][Atlas86]. Other systems focus on file replication [Tomlinson85][Ellis83], or transaction support [Cabrera87] [Wilkes80][Mitchell82] [Oki85], which are topics not considered in this dissertation.

2.6. The Sprite File System

There are three important aspects of the Sprite file system: the scale of the system, location-transparency, and distributed state. The effect of these features on the design of Sprite are discussed below.

2.6.1. Target Environment

The scale of the distributed system is important. There are roughly three classes of distributed systems: 1) small collections of autonomous hosts that wish to cooperate to a limited extent, 2) larger collections of hosts, usually personal workstations, that are more interdependent, and 3) very large collections of hosts that can span organizational boundaries. Sprite is targeted for the middle range, a medium sized network (up to 500 hosts) of high-performance diskless workstations and a set of supporting file server machines. A small example is shown in Figure 2-2. This moderate scale allows for certain simplifying assumptions. For example, server hosts can be located using broadcast. geographical distribution is not an issue, nor is per-host autonomy. Instead, Sprite unifies a

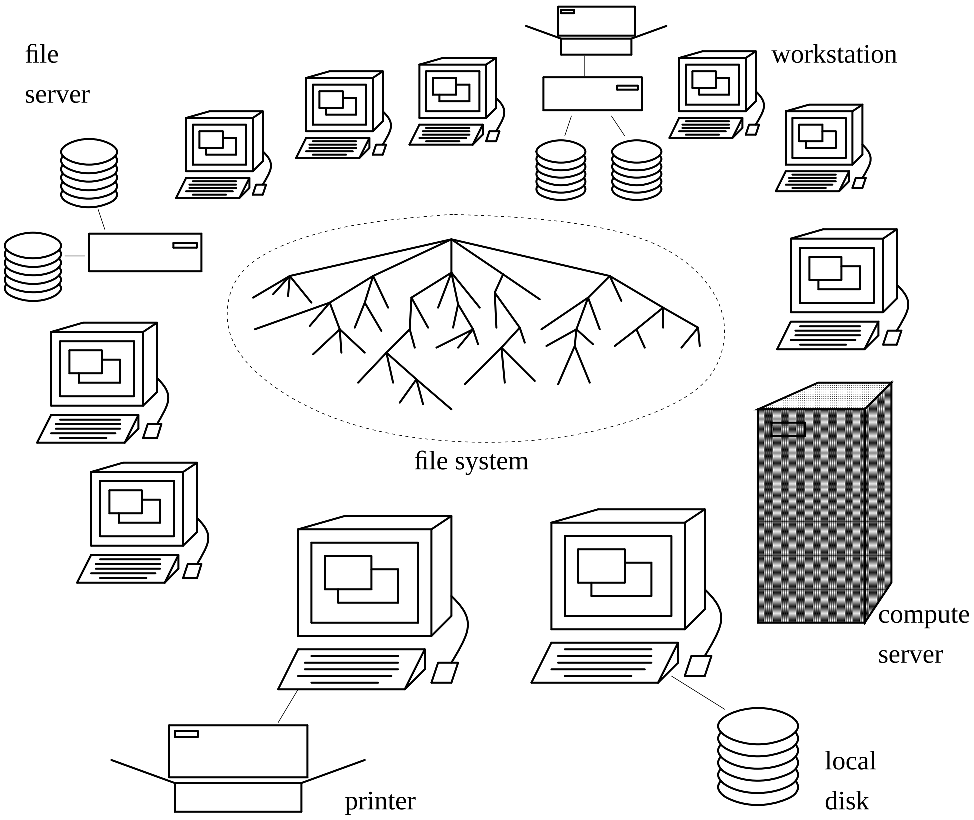


Figure 2-2. A typical Sprite network. The file system is shared uniformly by all hosts, and it has names for devices and user-level services as well as files. Most Sprite hosts are diskless workstations.

moderate size network into a single computing environment.

Sprite was developed on Sun-2 and Sun-3 workstations, which are rated around 1-2 MIPS (millions of instructions per second) processors. They have from 4 to 16 Mbytes of main memory. The file servers have one or more disks, each with a capacity of 70-1000 Mbytes. Today Sprite runs on the latest DEC and Sun workstations which have 10-20 MIP processors and 12-128 Mbytes of main memory. Our system currently has four file servers and about 40 clients, and it is gradually expanding.

2.6.2. Transparency

The Sprite file system is *location-transparent*: the same interface applies to file system objects regardless of their location in the network. Transparency begins with a naming system that does not reflect an object's location. The approach taken in Sprite is to extend the file system name space so it provides location-transparent names for devices and user-level server processes (i.e., pseudo-devices) as well as names for files and directories. This approach is in contrast to the approach of adding a separate name service that is used to locate file servers, device servers, and other user-implemented services. The benefit of extending the file system name space is that users and applications are presented with the same interface found in a stand-alone system. All the details about where objects are kept is hidden by the operating system.

Access to objects must be location-transparent, too. Operations on any type of object must apply if the object is on another host. Thus, all objects are available throughout the system. In contrast, most other distributed systems provide remote file access, but very few systems provide access to remote devices. Remote device access is important because devices like printers and tape drives can be shared by all clients. Displays can be accessed remotely to provide direct user-to-user communication. Disks can be accessed remotely for diagnostic and administrative reasons.

Remote device access is more difficult than remote file access, however, because many devices have indefinite service times. A file access, in contrast, completes or fails in a relatively short period of time, on the order of 10s of milliseconds. Remote operations have to be structured so that critical resources are not tied up for an indefinite period of time while waiting on a slow device. In RFS, for example, remote device operations that have to wait are aborted if server resources would be exhausted [Rifkin86]. In Sprite, remote operations are structured differently so that long-term waiting doesn't use up critical server resources.

2.6.3. Stateful vs. Stateless

The Sprite file system maintains internal state information that is itself distributed among the hosts in the network. There are two approaches to maintaining this state. In the so-called ‘‘stateless server’’ approach, each server host is basically independent of other hosts, and it can service requests with no state about previous requests. Any long-term state maintained by the service is kept in non-volatile memory so that the server can crash at any time and not corrupt itself. One of the first examples of a stateless service was WFS[Swinehart79]. The NFS[Sandberg85] protocol also uses the stateless server model. The main advantage of the stateless server approach is that failure recovery is simplified. The server merely reboots and performs some consistency checks on its non-volatile data structures. Clients do not need to take any special recovery action. They can retry their service requests until the server responds. However, there can be a performance cost in the stateless approach because the server must reflect all important state changes to its disk.

A more optimistic approach is to allow the server to build up volatile state about how its resources are currently being used. The extra state can be used to optimize access to the server's resources. Consider an I/O stream from a process on a client to an object on a server. If the server keeps state about an I/O stream then permission checking can be done when the I/O stream is created and need not be repeated each time the I/O stream is used. A stateless server, in contrast, must recheck permissions each time a file is read or written. A stateless server also has difficulty supporting the UNIX **unlink** operation. **Unlink** removes a directory entry for a file, but the file is not deleted from disk until I/O streams that reference the file are closed. A server cannot support this semantic of **unlink** if it does not keep state about I/O streams.

A more significant performance-related example concerns the main memory buffer cache of file data. A client of a stateful server can count on help from its server to keep its cache consistent with activity by other clients[Nelson88a] [Back87] [Howard88]. The server can inform Client A that its cached version of a particular file is no longer valid because Client B wrote a new version. A stateless server, in contrast, cannot help its clients in this fashion because it doesn't keep state about what clients are using its resources. Instead, clients of a stateless server must poll their server to determine if a cached file is up-to-date, and this reduces the efficiency of the client's cache. Furthermore, there can be periods of time between polling in which a client gets stale data from its cache[Sandberg85].

The main problem with the stateful approach, however, concerns failure recovery. There must be some way to recover the server's state if it fails, otherwise a server crash can cause clients to fail as well. As described in Chapter 6, Sprite relies on redundancy in the system to implement a state recovery protocol in which servers can clean up after failed clients and clients can help a failed server recover its state after it restarts.

2.7. Conclusion

The basic problem considered in this dissertation is how to organize a collection of computers connected by a network into a unified system. The approach taken in Sprite is to use the file system as the focal point of the system. This approach is motivated by timesharing systems where users shared all resources (including the CPU), and cooperative work was easy (albeit slow). By extending the file system to provide shared access to resources located throughout the network, the best features of the timesharing environment are regained along with the performance benefits of personal workstations. The UNIX file system, in particular, is chosen as a starting point because of its current popularity and because it embodies many good design decisions, including a hierarchical name space and the notion of device-independent I/O operations.

The latter half of this chapter considered related work and the various issues raised by distributed systems. The fundamental issues are naming and communication, which Sprite addresses by extending the file system name space to name other objects (e.g., devices and pseudo-devices), and by using a kernel-to-kernel RPC protocol for efficient remote access. A few key ideas were also discussed. The client-server model provides a way to structure the system. With a standard communication protocol (i.e. RPC) it is possible for a client to access any server in the network. Location transparency is important so that the distributed system has the look and feel of a unified system; details of the distributed environment can be managed by the operating system. The stateful nature of the system is also important. Server state is required to support all the semantics of a stand-alone UNIX file system and to support a high-performance caching system, but server state makes failure recovery more difficult.

CHAPTER 3

Distributed File System Architecture

3.1. Introduction

This chapter describes the internal software architecture of Sprite's distributed file system. The architecture supports the file system's goal of providing access to files, peripheral devices, and pseudo-devices that are located all around the network. These *objects* are all accessed via the same file system interface, regardless of their location in the network. The underlying distribution of the objects among a collection of file servers and client workstations is not important to users. A ‘‘single system image’’ is provided where any host is as usable as any other, much like a timesharing system where all the terminals provide equivalent access to the central host.

The architecture makes a fundamental distinction between naming and I/O operations. This distinction allows file servers to play the role of ‘‘name server’’ for file system objects that are not kept on the file servers. These objects include devices and pseudo-devices (i.e., user-level server processes). The architecture supports a three-party situation among the file server that holds the name for an object, the *I/O server* that implements I/O operations on the object, and the client that uses the object. Thus, the Sprite architecture provides the flexibility found in systems that introduce a network name service [Wilkes80] [Terry85] [Renesse89], but it does this by reusing the mechanisms needed for remote file access. By taking this approach, the architecture optimizes the common case of file access because there is only a single server involved, the file server.

The addition of devices and services to the file system creates problems that are not present in other distributed file systems. The main problem is that devices and arbitrary user-implemented services can have indefinite service times, while a file access has a bounded service time. It is important to structure the system so that critical operating system resources cannot be exhausted by too many indefinite service calls. For example, each Sprite kernel keeps a pool of kernel processes that are used to service remote requests. If the server processes were allowed to block indefinitely on a device, then all the server processes could be used up and prevent other requests from being serviced. Thus, chapter describes a synchronization technique for remote blocking I/O operations that solves this problem. The technique is also applicable to the **select** operation, which is used to wait on many different files, devices, and services simultaneously.

A distinctive feature of the Sprite file system is that it supports *stateful* servers. Sprite servers maintain state about how their resources are being used by clients. The state is useful for a number of reasons: 1) state helps guard against attempts to misuse resources, 2) state is needed for file locking and exclusive access modes, 3) state is used to implement remote blocking I/O operations, and 4) state helps the servers optimize access to their resources. In short, server state is required to fully implement all the semantics found in a stand-alone file system. The most significant example of this is the caching system used in Sprite. All Sprite hosts, clients and servers, keep main-memory caches of recently used file data to optimize file accesses [Nelson88a]. The file servers maintain state information describing how their files are cached. The servers guarantee that clients always have a consistent view of file data, even during concurrent updates by different clients. They rely on their state information to achieve this consistency. This chapter describes the way the system maintains its state efficiently during normal use, while Chapter 6 describes the way the system repairs its state after failures.

In contrast, the ‘‘stateless’’ approach taken by systems such as WFS[Swinehart79] and NFS[Sandberg85] limit the semantics of the system. A stateless server cannot make guarantees to clients that cache data about the consistency of their cached data. Instead, it is the client’s responsibility to poll the server. Also, the stateless approach can lead to garbage collection problems; often there is no way to know if names still exist for objects, or if objects still exist for names [Terry88]. In a stand-alone UNIX file system, for example, it is possible to remove the name for a file while there are still open I/O streams accessing the file. The file is not deleted from disk, however, until the I/O streams are closed. This requires shared state between the naming and I/O parts of the system that is found in few distributed systems. Thus, it is the combination of stateful servers and the use of the file system name space as the name service that distinguishes Sprite from other work in distributed systems.

A notable exclusion from this file system architecture is any description of the disk sub-system, file formats, disk allocation policies, etc. These are considered ‘‘object-specific’’ details that only pertain to file access They are hidden below the internal I/O interface. The current implementation of the Sprite disk sub-system is loosely based on the fast-file-system work of McKusick[McKusick84]. Current work with Sprite includes research into log-structured file systems to achieve even higher disk bandwidth [Ousterhout89a]. However, the details of the disk sub-system are not important in the discussion of this architecture, which mainly addresses issues raised by the network.

The rest of this chapter is organized as follows. Section 3.2 motivates the division of the architecture into general-purpose and object-specific modules. Section 3.3 describes the data structures that represent an I/O stream. Section 3.4 describes the internal naming interface, and Section 3.5 describes the internal I/O interface. Section 3.6 presents measurements of the system. Section 3.7 compares the Sprite file system architecture with UNIX-based architectures, and Section 3.8 concludes the chapter.

3.2. General Description

The file system is used to access many different types of objects. The goal of the architecture is to gracefully handle the many different cases that need to be implemented. It is important to carefully organize the architecture so that the combination of different types of objects and different features does not lead to an explosion of special case code in the implementation. The code is organized into general-purpose modules and object-specific modules. The general-purpose modules are independent of the type of the object being accessed, so their functionality can be applied to all types of objects. The object-specific modules hide the details about different cases that the architecture has to handle.

The architecture defines an interface between the general-purpose routines and lower-level, object-specific routines. There are several object-specific implementations of the interface (i.e., one for local files, one for remote devices, etc.). The general-purpose routines invoke the proper object-specific routines through the interface, and the actual case they invoke is hidden. There are no object-specific dependencies in the general-purpose routines. This approach has become known as an ‘‘object-oriented’’ approach, although the implementation of Sprite does not rely on an object-oriented language or other special support. Modularity has been achieved by careful design of the main internal interfaces and supporting data structures. The indirection through the interface to different object-specific implementations is achieved with a simple array of procedure variables that is indexed by a type. The type is explicit in the data structures that represent an I/O stream as described below in Section 3.3.

3.2.1. Naming vs. I/O

The system call interface to the file system reflects a basic split between naming and I/O operations. Figure 3-1 lists operations on pathnames. Figure 3-2 lists the operations made on open I/O streams. (For completeness, the remaining FS system calls are given in Figure 3-3.) The distinction between naming and I/O is important in the Sprite file system because naming operations are implemented by the file servers, while any host can be the I/O server for an object (i.e. a device). The distinction between naming and I/O is supported by having two main internal interfaces in the architecture, one for operations on pathnames and one for operations on open I/O streams. The naming interface hides the distribution of the name space among local and remote file servers, and it is largely independent of the type of object being named. The I/O interface hides the different kinds of objects accessed via the file system, and it is independent of how the name of an object is implemented. These interfaces are distinct so that different servers can implement the naming and I/O operations for the same object. Note, however, that the file servers are also the I/O servers for their files, and this allows optimization of file access in comparison with a system that uses an external name service. In Sprite, there is no need for a third party name service when accessing files.

The overall structure of the architecture is shown in Figure 3-4. The different cases underneath the naming interface handle local and remote pathnames. The different cases underneath the I/O interface handle files, pipes, devices, and remote access (all remote devices are treated identically on the client side). It is possible to name a remote device with a local pathname, or a local device with a remote pathname; all combinations are possible because the I/O and naming interfaces are independent.

Note that the term ‘‘object-specific’’ is used with both the naming and the I/O interfaces. With the naming interface, however, object-specific means ‘‘local pathname’’ as

System Calls on Pathnames

Fs_AttachDisk(*pathname*, *device*, *flags*)
(mount, umount) Mount or dismount a disk on a file server.

Fs_ChangeDir(*pathname*)
(chdir) Change a process's working directory.

Fs_CheckAccess(*pathname*)
(access) Verify access permissions on a pathname.

Fs_GetAttributes(*pathname*, *attributes*)
(stat) Return the attributes of an object.

Fs_SetAttributes(*pathname*, *attributes*)
(chmod, chown, utimes) Change the attributes of an object.

Fs_MakeDevice(*pathname*, *devAttrs*)
(mknod) Create a device file.

Fs_MakeDir(*pathname*)
(mkdir) Make a directory.

Fs_Remove(*pathname*)
(unlink) Remove a directory entry.

Fs_RemoveDir(*pathname*)
(rmdir) Remove a directory.

Fs_Rename(*pathname*, *newname*)
(rename) Change the name of an object from *pathname* to *newname*.

Fs_HardLink(*pathname*, *linkname*)
(link) Create another directory entry (*linkname*) for an existing object (*pathname*).

Fs_SymLink(*pathname*, *linkname*)
(symlink) Create a symbolic link (*linkname*) to another name (*pathname*).

Fs_ReadLink(*pathname*)
(readlink) Read the value of a symbolic link.

Fs_Open(*pathname*, *flags*, *mode*, *streamIDPtr*)
(open) Create an I/O stream given a pathname.

Figure 3-1. Sprite system calls that operate on pathnames. The equivalent UNIX system calls are given in parentheses.

System Calls on I/O Streams

Fs_GetNewID(*streamID, newID*)
(dup, dup2) Create a new stream ID for an existing I/O stream.

Fs_GetAttributesID(*streamID, attributes*)
(fstat) Return the attributes of an object.

Fs_SetAttributesID(*streamID, attributes*)
(fchmod, fchown) Change the attributes of an object.

Fs_Read(*streamID, buffer, numBytes*)
(read) Read data from an I/O stream.

Fs_Write(*streamID, buffer, numBytes*)
(write) Write data to an I/O stream.

Fs_IOControl(*streamID, cmd, inSize, inBuf, outSize, outBuf*)
(ioctl, trunc, ftrunc, flock, fsync, lseek) Do an object-specific function.

Fs_Select(*numStreams, readMask, writeMask, exceptMask, timeout*)
(select) Wait for any of several I/O streams to become ready for I/O, or until a timeout period has expired. The streams are specified by bitmasks where each bit corresponds to a stream ID.

Fs_Close(*streamID*)
(close) Close an I/O stream.

Figure 3-2. Sprite system calls that operate on I/O streams. The equivalent UNIX system calls are given in parentheses.

Miscellaneous FS System Calls

Fs_Command(*cmd, buffer, option*)
General hook used for testing, debugging, and setting kernel parameters.

Fs_SetDefPerm(*permissions*)
(umask) Set the default permissions for newly created files.

Fs_CreatePipe(*readStreamID, writeStreamID*)
(pipe) Create a pipe are return two streamIDs corresponding to the reading and writing ends of the pipe.

Figure 3-3. Miscellaneous Sprite system calls that pertain to the file system. The equivalent UNIX system calls are given in parentheses.

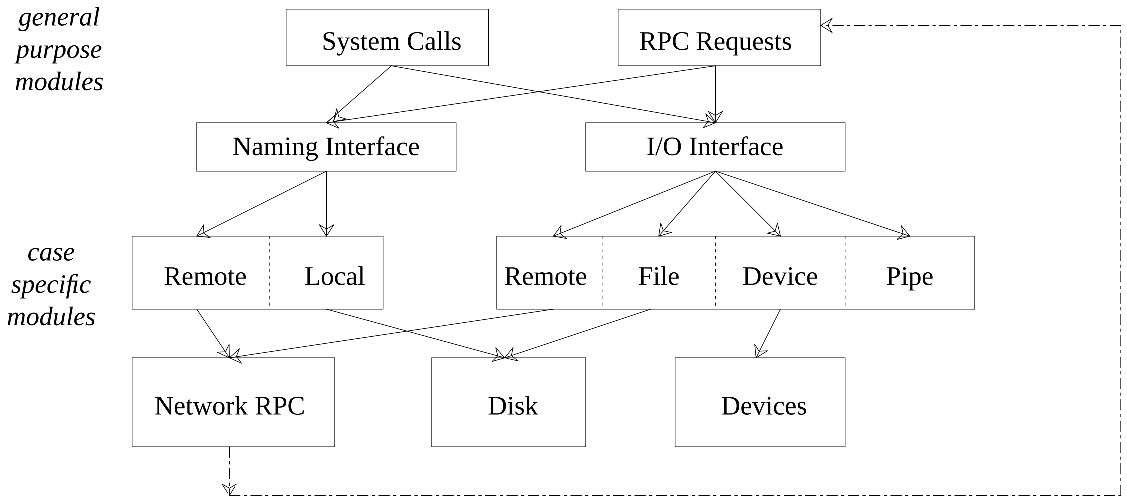


Figure 3-4. The modular organization of the file system architecture. There are two primary internal interfaces, one associated with operations on pathnames, and another associated with operations on I/O streams. General-purpose modules invoke object-specific modules through these interfaces, thus hiding the details associated with each case. Network requests are also dispatched through the internal interfaces in order to completely hide the use of RPC.

opposed to ‘‘remote pathname,’’ and the type of the object being named is not important. With the I/O interface object-specific means ‘‘local file,’’ ‘‘remote device,’’ etc.

3.2.2. Remote Procedure Call

One can view the architecture’s internal interfaces as an interface between client and server. The top-level routines above the interfaces are the ‘‘client’’ parts of the system, and the low-level routines are the ‘‘servers’’ for various kinds of objects supported by the system. With this viewpoint, the Remote Procedure Call (RPC) model provides a natural way for top-level client routines to invoke remote service procedures (i.e., for remote file or device access).

RPC fits into the naming and I/O interfaces as follows. In the remote case, the top-level, general-purpose naming and I/O routines invoke RPC stub procedures through the object-specific naming and I/O interfaces. These stubs package up their parameters into messages and use a network RPC protocol to invoke the corresponding object-specific procedure on the remote server. At the server, complementary RPC stub procedures call through the internal naming and I/O interfaces to invoke the correct service procedure. Note that the internal interface is used on both the client and the server to hide the details associated with network RPC. The service procedures are invoked in the same way whether they are called through the interface directly, or whether RPC has been used to forward the operation from a remote client. Similarly, the top-level routines are the same whether they invoke the service procedure directly through the interface, or whether RPC is used to forward the operation to the remote server.

Sprite uses a custom RPC protocol for kernel-to-kernel network communication[Welch86a]. The network protocol handles the problems of lost network messages by setting up timeouts and resending messages, if needed. Explicit acknowledgment packets are eliminated, if possible, by using the Birrell-Nelson [Birrell84] technique of implicit acknowledgments: a reply automatically acknowledges the corresponding request, and a new request serves as the acknowledgment for the previous reply. In most cases only two network messages are required, so there is low latency in the protocol. To increase throughput, large data transfers are supported by using multiple network packets. Finally, the Sprite RPC protocol guarantees that the remote procedure will be executed *at-most-once*; if a communication failure occurs then the caller cannot tell if the callee failed before or after execution of the remote procedure. Measurements of the RPC performance are presented in Section 3.6.1.

3.3. Internal Data Structures

The detailed description of the architecture begins in this section with the data structures that represent file system objects and the I/O streams between processes and objects. These data structures represent the distributed state of the system. With this information as a base, the internal naming and I/O interfaces are described in more detail in Sections 3.4 and 3.5.

3.3.1. Object Descriptors

The central data structure in the architecture is an *object descriptor*. Object descriptors are typed data structures that encapsulate the state of a particular object and are used during operations on the object. The contents of an object descriptor are initialized and manipulated only by the object-specific module that matches the type of the object descriptor. Bugs can be patched and features can be added for one type of object without breaking the implementation of other types of objects.

Each object descriptor begins with a *descriptor ID* that is composed of four fields. The first field is a type that is used to dispatch through the internal I/O interface to different object-specific procedures. The types are given in Table 3-1. They correspond to the different cases that the I/O interface has to accommodate. The second field in a descriptor ID identifies the I/O server for the object. The explicit server ID facilitates the use of RPC, and it also ensures that different I/O servers do not define the same descriptor IDs. The third and fourth fields are for use by the I/O server to distinguish the object descriptor from others of the same type.[8]

---

[8] Within the Sprite code the term ''I/O handle'' is used instead of ''object descriptor''. Object descriptors type definitions have names like ''Fsio_FileIOHandle'', ''Fsio_DeviceIOHandle'', and ''Fsrmt_IOHandle''.

Object Descriptor Types	
LOCAL_FILE	A file on a local disk.
REMOTE_FILE	A file on a remote disk.
LOCAL_DEVICE	A local device.
REMOTE_DEVICE	A device on a remote host.
LOCAL_PDEV	A locally executing user-level server process.
REMOTE_PDEV	A remote user-level server process.
LOCAL_PIPE	A one-way byte stream between processes.
REMOTE_PIPE	A pipe that is remote because of process migration.

Table 3-1. Object descriptor types used by the operating system. The corresponding LOCAL and REMOTE types are used on the I/O server and remote clients, respectively.

The I/O server chooses its object descriptor IDs so that they are the always the same for a particular object. Instead of being random UIDs, they are based on some property of the object. Device servers, for example, embed the device type and device unit in a device's object descriptor ID. File servers put a disk number and file number into a file's object descriptor ID. The deterministic choice of an object descriptor ID is important for two reasons. First, the recovery protocol described in Chapter 6 depends on this so that clients can help server's rebuild their internal state, which includes the contents of object descriptors. Second, it is possible for a device to have more than one name (this is described below in Section 3.4.1.2). By choosing the device's object descriptor ID based on the device type and unit number, the system ensures there is only one object descriptor for the device, even if the device has multiple names.

An important property of the object descriptors used in Sprite is that there are complementary LOCAL and REMOTE object descriptors for use on the I/O server and client, respectively. This distinction supports a clean separation between object-specific procedures used by remote clients and those used on the I/O server. The server's object-specific procedures are only concerned with accessing the underlying object; they are not littered with special cases against a remote object. The client's object-specific procedures are concerned with efficiently forwarding the operation to the remote I/O server. Note also that if the client and the I/O server are the same host, then only the LOCAL object descriptor is used, and the local service procedures are accessed directly through the I/O interface.

The association between the object descriptors on the client and the I/O server is defined by the IDs of their corresponding object descriptors. The object descriptor IDs differ only in their type and the remaining three fields are the same. This correspondence means that clients pass descriptor IDs (not the whole descriptor) to the server, and the server can locate its corresponding descriptor by converting the client's type to a server type (e.g., REMOTE_FILE to LOCAL_FILE). The conversion is done by general purpose code on the server so that RPC stub procedures can be shared. The stubs take care of locating the server's descriptor and then they branch through the internal I/O interface to object-specific service procedures.

Code sharing among RPC stubs on the client is also possible because they are invoked through the internal I/O interface. The REMOTE_DEVICE, REMOTE_PDEV, and REMOTE_PIPE implementations all share the same RPC stub procedures. These stubs are ultimately used by the REMOTE_FILE implementation, too, to read and write data to and from the cache.

To summarize, an object descriptor encapsulates the information needed to access an object. The type of the object descriptor indicates the particular case that applies to the object, and this type is used to branch to object-specific implementations of the I/O interface. The descriptors are tailored towards the different needs of the client and the server. The I/O server keeps LOCAL object descriptors and its object-specific implementations access the underlying object directly. The remote clients keep REMOTE descriptors and use RPC to forward operations to the I/O server, although object-specific optimizations for the remote case are supported (e.g., use of the data cache during remote file accesses). Object descriptor IDs are passed between clients and servers, and the RPC stubs rely on a well-defined mapping between LOCAL and REMOTE descriptor types in order to find their corresponding descriptor.

3.3.2. Stream Descriptors

An I/O stream between a process and an object is represented by the data structures shown in Figure 3-5. For each process, the kernel keeps a *stream table* that contains pointers to *stream descriptors*. Each stream descriptor references an object descriptor. A stream descriptor is created by the Fs_Open system call, and a pointer to this descriptor is put into the stream table. The index of the pointer in the stream table is returned to the process as its handle on the I/O stream.

The stream descriptor provides a level of indirection between the process and the object descriptor that is useful for three reasons. First, different processes may be using an object in different ways (e.g., one for reading and one for writing). The intended (and authorized) use of a stream is recorded in the stream descriptor and used to guard against unauthorized use of the object. Second, processes may share an I/O stream as the result of stream inheritance. A stream descriptor is created during the Fs_Open system call, and this stream descriptor becomes shared when the stream is inherited by another process. A reference count in the stream descriptor reflects this sharing. Third, sequential access to an object is supported by maintaining the current access position in the stream descriptor. Read and write operations advance the access position so that consecutive operations on an I/O stream operate on sequential ranges of an object.

The Sprite data structures for an I/O stream are similar to those in UNIX[9] except for

---

[9] UNIX terminology is different, however. Instead of ‘‘stream table’’ UNIX uses ‘‘open file table.’’ Instead of ‘‘stream descriptor’’ UNIX uses ‘‘open file descriptor.’’ Instead of ‘‘object descriptor’’ UNIX uses ‘‘inode.’’ The Sprite architecture uses ‘‘stream’’ instead of ‘‘open file’’ to reflect that I/O streams can be connected to many different kinds of objects.

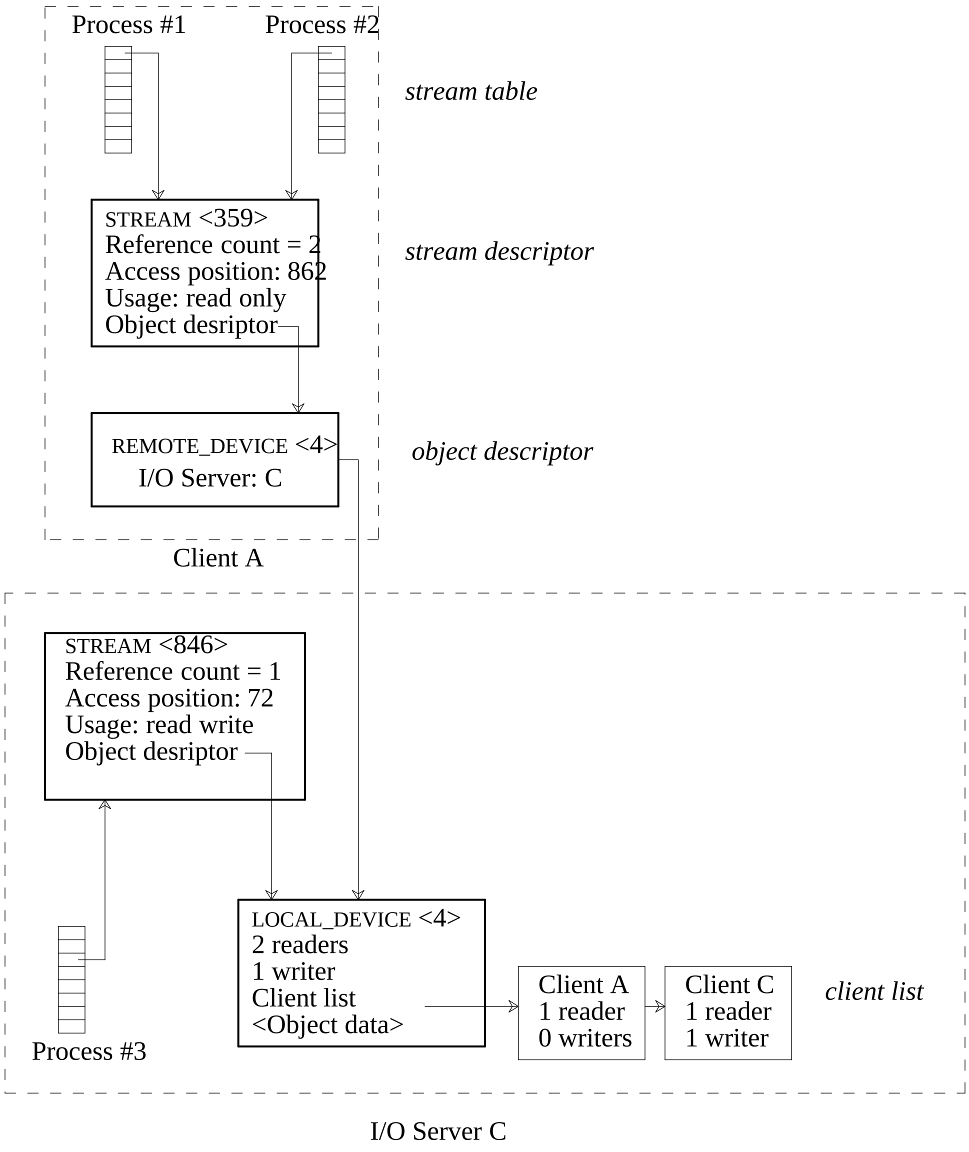


Figure 3-5. The data structures that represent an I/O stream from a process to an object. There are two levels of sharing possible. First, there may be many distinct streams to the same object. Secondly, a stream may be shared by more than one process. In this example there are three processes accessing the same device. Two of the processes are sharing one stream, and a third is on the I/O server. The I/O server keeps a client list with the object descriptor that records how different clients are using the object. Note that the client list counts the number stream descriptors on each client, which can be different than the number of processes using the streams due to stream sharing.

two extensions related to remote clients. First, Sprite has corresponding LOCAL and

REMOTE object descriptors as described above. Secondly, each LOCAL object descriptor has a *client list* that the server uses to keep state about how its object is being used by remote clients. Each client list entry counts the readable, writable, and executable I/O streams that originate from each client, as well as information about file locks held by the client. The per-client organization of this information is required for three reasons. First, the Sprite file caching system has to know what clients are reading and writing files to guarantee that clients see a consistent view of file data [Nelson88a]. Second, servers disallow modification of files being used for an executable program image. Third, if a client crashes then the I/O server can clean up after it by simulating **close** operations on I/O streams from that client and releasing any file locks that it held.

3.3.3. Maintaining State About I/O Streams

This section describes the stream accounting information that the system keeps so that the I/O server has an accurate view of the I/O streams to its objects. The accounting information includes a *reference count* on stream descriptors, and *usage counts* in the client list that count up the I/O streams used for reading, writing, and execution. These are maintained according to the following invariants, which refer to Figure 3-5 for examples.[10]

- The reference count on a stream descriptor reflects use of the stream by local processes. Stream #359 is shared by two processes on Client A, for example, so it has a reference count of 2.
- Each client list entry counts the readable, writable, and executable I/O streams that originate from that client. If a stream is used for more than one purpose (e.g., reading and writing), then it is counted once for each use. The LOCAL_DEVICE object descriptor in Figure 3-5 has two client list entries corresponding to the streams on client A and on itself.
- There is a set of summary usage counts in a LOCAL object descriptor that represents the total number of streams to the object, both local streams and remote streams. This can be computed from the client list, but it is maintained separately in order to simplify conflict checking and garbage collection of unused descriptors.

An important aspect of these data structures is that sharing of streams is not visible at the object descriptor level; if a stream is shared among processes, then it still only counts as one stream at the object level. Each process that is sharing the stream has the same access permissions (e.g., reading and/or writing) so it does not matter how many different processes share the stream. It is important to hide stream sharing from the object level because it eliminates the need to communicate with the I/O server when new processes share a stream. Otherwise, if the I/O server had to know about stream sharing, then every process creation would suffer the cost of notifying the I/O server about new sharers of a

[10] File lock information is also kept in the client list. It includes an exclusive lock bit and a count of shared lock holders.

stream.

The basic stream operations that affect the stream accounting state are described below.

Fs_Open — Fs_Open creates a stream descriptor with reference count equal to 1. The object descriptor referenced by the stream has to be initialized or have its usage counts incremented to reflect the new stream. If the stream is to a REMOTE object descriptor then the corresponding LOCAL object descriptor, and its client list, has to be updated.

Proc_Fork/Fs_GetNewID — These operations increment the reference count on a stream descriptor. Proc_Fork creates a child process that shares all its parent's streams. Fs_GetNewID adds an additional reference from a process's stream table and returns a new handle (ID) for the stream. In either case, there is no new stream descriptor so there is no need to change the stream usage accounting on the object descriptor.

Fs_Close — This decrements the reference count on the stream. If it is the last reference then the stream descriptor will be destroyed. In this case the stream usage information in the object descriptor has to be decremented.

These operations imply that the I/O server sees every Fs_Open operation so that it can keep track of all the I/O streams. Also, for each Fs_Open the I/O server will see one Fs_Close. This is the fundamental difference between a stateful and a stateless system. A stateless protocol such as NFS has no **open** and **close** operations, while a stateful protocol includes **open** and **close** so the server can maintain state about connections (I/O streams) to its resources.

3.4. The Naming Interface

The internal naming interface is described in this section. The operations are similar to those found in other remote file systems. The novel aspects of the naming interface mainly concern the way the name space is distributed among servers, and this is the topic of Chapter 4. In this chapter it is sufficient to understand that a server exports a directory in the global file system hierarchy to the rest of the network, and this causes all lookups below that point in the hierarchy to be directed to the server. Chapter 4 describes the distributed name resolution mechanism in detail. The naming interface is described here because it has some effect on the I/O interface described below.

There are two implementations of the naming interface: one for remote pathnames and one for local pathnames. A client branches to the correct implementation after mapping the pathname argument of a system call into a *server token* and a relative pathname. The server token is a descriptor ID for a directory exported by the server. The token identifies the server and its type to the client, and the client uses the type to dispatch through the internal naming interface. The server begins resolving the relative pathname at the directory identified by the server token.

The operations that compose the internal naming interface are described below.

NAME_OPEN(*token, path, useflags, userIDs, type, permissions, results*)
: This operation is the first half of creating an I/O stream. The *useflags* parameter indicates the manner in which the I/O stream will be used, whether the object should be created if it does not exist, and other details that are not important here. (A complete definition of flag bits can be found in Table D-5 in Appendix D.) The *type* parameter is used to constrain the open to match a particular type (e.g., directory or pseudo-device), but a default value indicates that any type is acceptable. (Types are defined in Table D-4 in Appendix D.) The *userIDs* authenticate the user to the file server. The *permissions* define the access control on newly created objects. NAME_OPEN returns *results,* which contains a descriptor ID and any other information needed to complete the setup of the I/O stream. These are passed through the IO_OPEN entry point in the I/O interface as described in Section 3.5.

GET_ATTRIBUTES(*token, path, userIDs, attributes*)
: This returns the attributes of an object. The *userIDs* argument is used for authentication, and the object's attributes are returned in the *attributes* parameter. This is distinct from NAME_OPEN because NAME_OPEN includes some processing associated with creating an I/O stream as explained below in Section 3.4.1.

SET_ATTRIBUTES(*token, path, userIDs, attributes, flags*)
: This changes the attributes of an object. Different attributes (access permissions, ownership, etc.) are changed in different situations, and the *flags* parameter contains bits that indicate which fields of the *attributes* parameter should be applied to the object's attributes.

MAKE_DEVICE(*token, path, userIDs, devattrs*)
: Create a special file that represents a device. The *devattrs* parameter specifies the device type, unit number, and the I/O server. Section 3.4.1.2 describes a special ‘‘localhost’’ value that can be used for a device's I/O server. This maps the device file to the instance of the device on the client.

MAKE_DIR(*token, path, userIDs, permissions*)
: Create a directory with the given permissions.

REMOVE(*token, path, userIDs*)
: Remove an entry from a directory. This applies to all objects except directories. An important side effect of removing a directory entry for regular files is that files are removed from disk after all names that reference them have been removed (see HARD_LINK).

REMOVE_DIR(*token, path, userIDs*)
: Remove a directory. It must be empty for this to succeed.

HARD_LINK(*token1, path1, token2, path2, userIDs*)
: Create an additional name (*path2*) for an existing object (*path1*). This creates a new directory entry that references the existing object.

RENAME(*token1, path1, token2, path2, userIDs*)
: Change *path1* to *path2*. This is implemented on the file servers by first making a

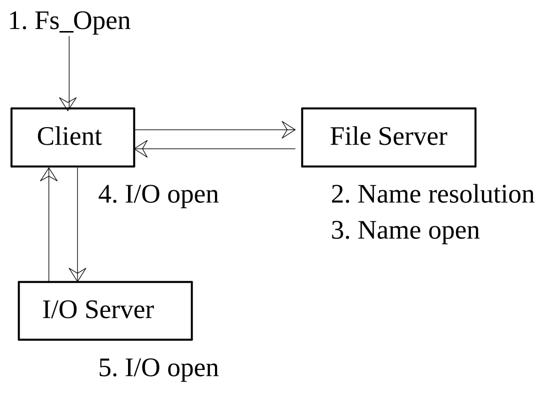
hard link (*path2*) and then removing the original name (*path1*). RENAME is explicit in the naming interface so that the servers can implement this sequence atomically.

SYM_LINK(*token1*, *path1*, *token2, path2*, *userIDs*)
Create a symbolic link (*path1*) to another pathname (*path2*).

Note that there is implicit context in the system call interface that is made explicit in the internal naming interface. The implicit context includes user authentication information, for example, and it also includes the token that identifies the server. It is important to make all information explicit so the naming operations can be passed around to remote servers.

3.4.1. Structure of the Open Operation

The Fs_Open operation, which creates an I/O stream, is divided into two parts, NAME_OPEN and IO_OPEN, to permit different servers for the naming and I/O interfaces. This is shown in Figure 3-6. However, to optimize the important case of opening regular files, in which case there is only a single server, the NAME_OPEN procedure is allowed to do some object-specific setup for the I/O stream that gets created by the IO_OPEN procedure. (IO_OPEN is the initialization procedure in the object-specific I/O interface, and it is described in more detail in Section 3.5.) This distinguishes NAME_OPEN from GET_ATTRIBUTES, which just returns attributes of the object and has no side-effects. Enough processing is done during NAME_OPEN on a regular file so




```
Fs_Open(pathname, arguments) {

    name_open(pathname, arguments)
        returns descriptorID, objectData

    io_open(descriptorID, objectData)
        returns descriptor

} returns I/O stream
```

Figure 3-6. The flow of control during an Fs_Open operation. The arrows indicate invocation of the name resolution and the I/O open phases on the file server and the I/O server, respectively. In the most general case these operations are invoked using network RPC.

the client does not have to contact the file server a second time during the IO_OPEN phase. This is an example of how using the file system name space as the name service provides an advantage over an external name service.

There is special-case processing that can be done during NAME_OPEN for other types of objects as well. This is performed by a *name open* procedure that is invoked on the file server after it has resolved a pathname to a particular file and obtained its LOCAL_FILE descriptor. There are currently three implementations of the name open procedure, one for files, devices, and pseudo-devices. The proper procedure is invoked based on a file-type attribute kept in the LOCAL_FILE object descriptor. The name open procedure is passed the LOCAL_FILE object descriptor and an indication of how the client will use the object. The name open procedure examines the attributes of the file to determine the proper object descriptor ID *for the client* and any other information needed during the IO_OPEN operation. The descriptor ID indicates the I/O server and the specific case that applies to the client (e.g., LOCAL_DEVICE, REMOTE_DEVICE, REMOTE_FILE, etc.). Thus, the name open procedure maps a file into a descriptor ID for the underlying object, and it extracts any object-specific attributes needed to create an I/O stream to the object. This is an example of the tight coupling between the naming and I/O halves of the file system.

3.4.1.1. Opening Files

The main job of the FileNameOpen procedure is to invoke the cache consistency mechanism. The cache consistency routines compare the intended use of the file with existing uses of the file that are recorded in the LOCAL_FILE descriptor client list. Cache control messages are issued as needed, and FileNameOpen completes after other clients complete their consistency actions (e.g., writing back dirty data to the server). FileNameOpen updates the client list to reflect the new client so there is no need for the client to contact the file server (again) during the IO_OPEN operation. The client only has to create (or update) its own REMOTE_FILE descriptor.

3.4.1.2. Opening Devices

The main job of the DeviceNameOpen procedure is to determine the case (local or remote) that applies to the client. Each device file has an attribute that identifies the I/O server for the device, and the relative location of the client and the I/O server determines if client's descriptor ID should be of type REMOTE_DEVICE or LOCAL_DEVICE. Note that these descriptors are distinct from the LOCAL_FILE object descriptor associated with the device file that names the device. These different descriptors are shown in Figure 3-7. There has to be a distinct descriptor for a device and its name because devices can be located on any host, not just the file server that keeps their name (the device file). It is also possible for more than one device file to name the same underlying device, and the use of different object descriptors for the name and the device means that there is still only a single LOCAL_DEVICE descriptor for a device.

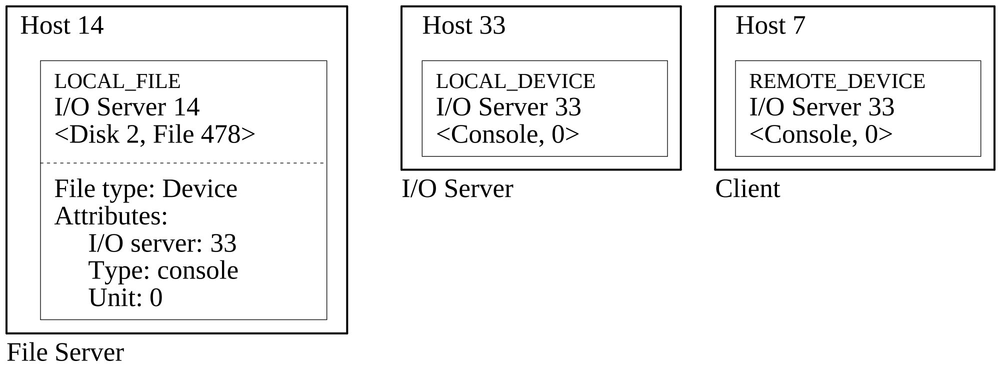


Figure 3-7. The object descriptors involved with remote device access. The file server resolves a name to an object descriptor for a LOCAL_FILE. In this case the file is a placeholder for a device, and the file attributes indicate the I/O server and the type of the device. The client sets up a REMOTE_DEVICE object descriptor, the I/O server sets up a LOCAL_DEVICE object descriptor.

There are two potential problems with device files in a distributed system. The first is that programs developed for a stand-alone system expect the device files in the ‘‘/dev’’ directory to be for local devices, but the ‘‘/dev’’ directory is shared by all hosts. The second problem is that if every device on every host requires a device file then there can be are large number of device files that are tedious to manage. These two problems are addressed in Sprite by the addition of special ‘‘localhost’’ device files that correspond to the local instance of a device. These files can be shared by all hosts, which reduces the number of device files needed in a large system.

The ‘‘localhost’’ device files are trivially implemented by using a distinguished value of the I/O server ID to indicate ‘‘localhost’’. This special value has the effect of mapping the device file to the client’s instance of the device. If the opening process has migrated, then its original host is used as the I/O server so that process migration remains transparent. Thus, there are two kinds of device files in Sprite: one has a specific host for the I/O server attribute, and the other has a special ‘‘localhost’’ I/O server attribute.

3.5. The I/O Interface

The interface to the I/O operations is based on object descriptors. The type in the object descriptor ID is used to branch to the correct instance of the I/O operation, and the object descriptor is passed through the I/O interface. The I/O interface is similar to those in other systems, except for the way blocking I/O is done and the way that object attributes are handled. These issues are discussed after the interface is described.

IO_OPEN(*descriptorID, objectData, useflags, descriptor*)

The IO_OPEN procedure completes the setup of an object descriptor. The *descriptorID* and *objectData* parameters are the results of a NAME_OPEN operation. The *useflags* indicate how the object will be used, i.e. for reading and/or writing. The result of IO_OPEN is *descriptor,* which has been initialized and is ready for I/O. In the case of a remote client, IO_OPEN may involve an RPC to the I/O server so it can initialize (or update) its object descriptor. As mentioned above, this RPC is optimized away for the REMOTE_FILE case.

CLOSE(*descriptor, useflags*)

End use of an object. This is called when an I/O stream is being destroyed. There may be many I/O streams active to the same object descriptor, and CLOSE just represents the destruction of a single stream. The *useflags* parameter indicates the manner in which the I/O stream was being used.

CLIENT_VERIFY(*clientDescriptorID, clientID, serverDescriptor*)

The CLIENT_VERIFY routine is used on each RPC, and it has two main purposes. The first is to convert from a client's object descriptor ID to the server's corresponding object descriptor. Conversion is done by mapping the type in the client's descriptor ID to the corresponding server-side type (e.g., from REMOTE_FILE to LOCAL_FILE) and then looking up the server's descriptor based on the new object descriptor ID. The second function of this procedure is to verify that the client is known to the server. Verification is done by checking the client list on the server's descriptor.

READ(*descriptor, processID, offset, buffer, count, signal*)

Read data from an object. The input parameters indicate a byte offset, byte count, a buffer for the data, and the client process's ID. The *count* parameter is modified to indicate how much data, if any, was read. READ returns a status code as its function value. If EWOULDBLOCK is returned then it indicates that no more data is available from the object at this time (some data may have been returned). In this case *processID* has to be saved by the I/O server for use in a notification message after the object has data ready. (This is discussed in more detail below.) End-of-file conditions are indicated by a SUCCESS status and zero bytes returned. Some objects cause software signals to be generated in exceptional conditions. This can be achieved by returning GEN_ABORTED_BY_SIGNAL and a signal value.

WRITE(*descriptor, processID, offset, buffer, count, signal*)

Write data to an object. The input and result parameters are the same as for READ. This writes zero or more bytes, up to the amount specified by *count*. Upon return, *count* is modified to indicate how much data was accepted by the underlying object. If a short count is returned then more WRITE operations are made until all the user's data is transferred, unless EWOULDBLOCK is returned. EWOULDBLOCK is handled as described for the READ operation. GEN_ABORTED_BY_SIGNAL and a signal value can also be returned.

IOCTL(*descriptor, cmd, processID, format, inBuf, inSize, outBuf, outSize, signal*)

Perform a special operation (*cmd*) on an object. A generic set of IOCTL commands are defined by the system, and the object implementation is free to define any other

IOCTL commands it needs. The *processID* is available so the I/O server can implement blocking operations by returning EWOULDBLOCK and saving the processID for later notification. A signal can also be returned in order to give the I/O server complete flexibility. The size and contents of the input and output buffers are dependent on *cmd*. The *format* parameter indicates the byte-ordering and alignment format of the client host. The IOCTL procedure has to fix up the input and output buffers if the I/O server has a different format than the client. This issue is discussed in more detail in Section 3.5.2.

SELECT(*descriptor, events, processID*)
: Poll an object for readability, writability, and exceptional conditions. The *events* parameter is an input and output parameter. On entry it indicates which of the three conditions are of interest, and on exit it indicates which of the conditions currently apply to the object. If the object is not ready for one of the requested conditions then *processID* is saved for notification when the object changes to a ready state.

IO_GET_ATTR(*descriptor, attributes*)
: Get the attributes of an object. The *attributes* is both an input and result parameter. On input it contains the attributes as recorded by the file server. IO_GET_ATTR can update any attributes (i.e. the access and modify times) for which it has more current values.

IO_SET_ATTR(*descriptor, attributes*)
: Set the attributes of an object. As with IO_GET_ATTR, the file server has already been contacted to change the attributes on long-term storage. This routine updates whatever attributes the I/O server maintains while an object is in use, typically the access and modify times.

3.5.1. Blocking I/O

Remote device access introduces a waiting problem that is not present with remote files. Files have bounded access times so it is feasible to block the service procedure on the file server until the I/O operation completes. Devices, pipes, and pseudo-devices, in contrast, have indefinite access times. Blocking the service procedure indefinitely at the I/O server would tie up RPC server processes, and these might be a limited resource. Interrupting the I/O operation would be complicated because the remote RPC server would have to be interrupted. The Fs_Select operation adds a further complication. The streams being selected may be connected to objects located all around the network. In this case it would not be appropriate to wait at the I/O server for any particular I/O stream because some other I/O stream may become ready first.

These problems can be avoided by returning control to the client if an I/O operation (or SELECT) would block. A special return code, EWOULDBLOCK, is returned to indicate this situation. This return code is interpreted by general-purpose routines which block the process in this case. A callback from the I/O server is required when its object is finally ready for I/O. In terms of the architecture, this approach means that the object-specific I/O routines do not block indefinitely. The only action taken by the object-specific module during a blocking I/O is to record the client process's ID so it can be notified later. The process is then blocked in the top-level Fs_Read, Fs_Write, or Fs_Select procedure. These top-level procedures retry their calls through the I/O interface after a notification message arrives from the I/O server.

The implementation of this blocking approach has to guard against a race condition between a process's decision to wait and notification messages from the I/O servers. The message traffic during a remote wait is shown in Figure 3-8. The notification message could arrive before the reply with the EWOULDBLOCK error code. If the early notification

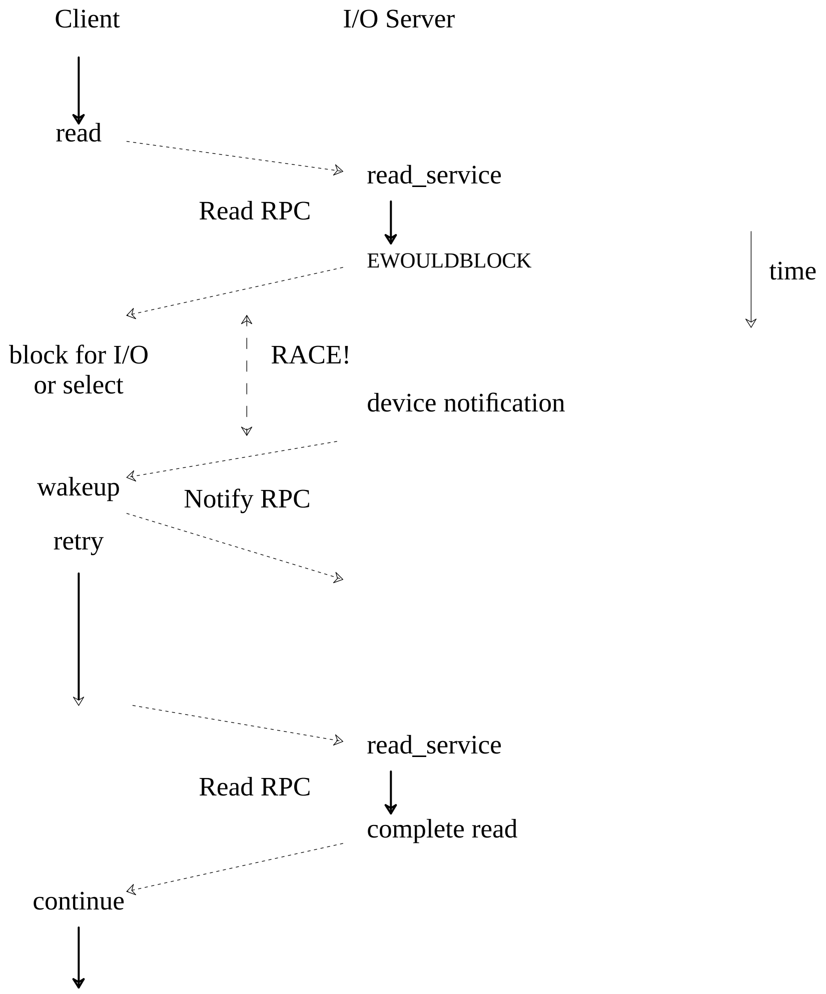


Figure 3-8. Message traffic during a blocking I/O. Control is returned to the client if an I/O operation would block. There is a potential race condition between the EWOULDBLOCK reply message and the notification message from the I/O server.

were ignored then the client might block forever. The race can occur during Fs_Select if an I/O server notifies the client while it is still polling other I/O servers. The race can occur during Fs_Read and Fs_Write if network messages are lost or reordered.

A simple algorithm is used to detect early notifications. Each process's state includes a *notify bit* that is cleared before the process begins its decision to block. The notify bit is set when a notification arrives. When a process finally decides it ought to wait, the kernel's waiting primitives check for the notify bit. If it is set it indicates that a notification has arrived early, and the process is not blocked. The process clears its notify bit and retries its operation. The retry loop is shown in Figure 3-9. A form of this loop is used in the top-level procedures Fs_Read, Fs_Write, Fs_IOControl, and Fs_Select.

3.5.2. Data Format

Sprite supports a heterogeneous network, one where clients and servers can be of different machine architectures. The main problem this causes concerns data representation, or byte-ordering. There are, for example, two main ways to arrange a four-byte integer in memory. There are also alignment policies that affect the layout of more complex data structures. When clients and servers with different data formats communicate,

```
done = FALSE
while (done == FALSE) {
    /* begin critical section */
    ClearNotifyBit();
    /* end critical section */

    status = BlockingOperation();

    /* begin critical section */
    if ((status == EWOULDBLOCK) AND NotifyBitIsClear()) {
        WaitForNotication();
    } else {
        done = TRUE
    }
    /* end critical section */
}
```

Figure 3-9. The basic structure of a wait loop. Manipulation of the notify bit and blocking the process is done inside a critical section in order to synchronize with notification messages.

some measures need to be taken to resolve these differences. One approach is to always convert every network message into a canonical format. This conversion is done in SunOS with its XDR (eXternal Data Representation) protocol. However, this approach adds overhead when the client and server have the same data format.

Sprite limits the format of network messages to make data format conversion easy, and it only does conversion when needed. This approach can be taken because network communication is done by the kernel using a small set of RPCs. RPC messages are divided into a header and two data parts. The first data part can contain an array of integers, while the second data part contains an uninterpreted block of data. The first part is sufficient for the simple data structures passed among kernels, and the second part is used for character strings (i.e. file names) and raw file data. The Sprite RPC protocol automatically reformats the first data part of network messages if the client and server have different formats. This case is detected by examining a header field with a well-known value. If the value is in the wrong format, then the packet is reformated.

The only case where this approach does not work is with the input and output buffers used in IOCTL. A device or service is free to define new IOCTL commands and their associated inputs and outputs. The RPC system does not know the format of the input and output buffers. This data is put into the unformated part of network messages and the data format conversion is left up to the IOCTL procedure.

3.5.3. Object Attributes

While most attributes are kept by the file servers, the I/O server for a device updates the access and modify times while a device is in use. Instead of having the I/O server constantly push these attributes back to the file server, the client contacts both the file server and the I/O server during Fs_GetAttributes and Fs_SetAttributes. The client invokes the file server via the GET_ATTRIBUTES or SET_ATTRIBUTES operation to manipulate the attributes kept there. These routines also invoke the object-specific name open procedure (i.e. FileNameOpen or DeviceNameOpen) to compute a descriptor ID which is returned to the client. The client uses the type of the descriptor ID to invoke an IO_GET_ATTRS or IO_SET_ATTRS procedure to complete the operation with the I/O server.

Attributes can be manipulated via an I/O stream to the object as well as by an object's name. Both cases are structured the same way: first the file server is contacted to get most of the attributes, and then the I/O server is contacted to get the current access and modify times. When an I/O stream is closed the I/O server pushes its versions of the access and modify times back to the file server for long term storage. In order to contact the file server the LOCAL_FILE descriptor ID corresponding to the name must be known to the client and the I/O server. This *name ID* is one of the results of the NAME_OPEN procedure. The client saves the name ID in its stream descriptor and the I/O server saves it in its object descriptor. (In the current implementation a device I/O server does not really save the name ID nor does it push the access and modify times back to the file server, but it should. This omission means that access and modify times of devices are only correct when they are open.)

Remote files have a slightly different problem regarding attributes. Caching by remote clients means that the file server may not have the most up-to-date access and modify times. However, instead of having one client contact all the other clients that cache the file, this is done by the file server because it can consult the file's client list to see what clients should be contacted. Currently the file server avoids fetching the access time of a file that is being executed because these files are heavily shared and it is not possible to determine exactly when a program image is referenced; the access time of an actively executed process is reported as ''now''. In the future we may avoid fetching access times from other clients altogether because this is a rarely used attribute. The modify time, on the other hand, is much more important, and delayed write caching on the clients means that the server may not have the most recent modify dates. However, in this case there is only one client that can be caching dirty data (this is a property of the Sprite cache consistency scheme[Nelson88a]) so the server only has to contact one other client.

An alternative implementation for remote devices would be to have clients maintain modify and access times instead of the I/O server. This approach is similar to what currently happens with regular files. If this alternative were implemented then clients would only contact the file server during attribute operations. The file server would make callbacks to the other clients using the object to get the current access and modify times. This approach adds extra state to the file servers. In the current implementation, for example, the file servers do not record which clients are using a remote device. The device's I/O server records this information instead.

3.6. Measurements

This section of the chapter presents measurements of the RPC performance and the RPC traffic in the system, and it gives a size breakdown of the various modules in the file system implementation.

3.6.1. RPC Performance

Figure 3-10 shows the performance of the Sprite RPC protocol when sending data blocks of various sizes from one kernel to another. The base cost of an RPC is higher on a Sun-3 because it has a slower CPU (i.e., 2 MIPS as opposed to 12-13 MIPS). The latency of a null RPC is about 1 msec on the DECstation and SparcStations, and about 2.45 msec on the Sun-3/75 workstation. This is about the same as the time it takes to switch back and forth between two user processes on the same host, so remote kernel operations are not too expensive. The incremental cost of sending more data in an RPC is not much different between machine types, which indicates that the time spent in the network interface hardware and on the network itself becomes dominant. The incremental cost for the DECstations is higher because the network interface does not have direct-memory-access (DMA). It takes additional CPU cycles to copy data in and out of the network interface. Large bandwidths are achieved by sending large data blocks. Multiple packets are used to ''blast'' the data block to the receiver, which returns a single packet in response [Zwaenepoel85]. Multiple packets are reflected by jumps in the graph

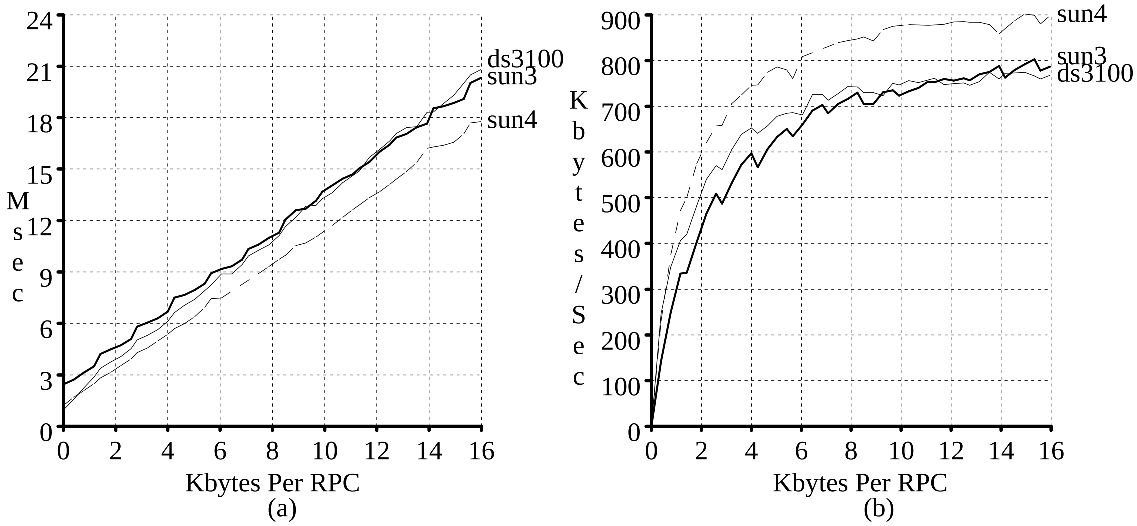


Figure 3-10. Performance of the Sprite kernel-to-kernel RPC network protocol on a 10 Mbit Ethernet. Graph (a) gives the time to transfer a data block from one kernel to another. Graph (b) shows the corresponding data bandwidths. The jumps in the graph reflect the use of additional network packets to transfer the data. The ‘‘sun3’’ line is for Sun-3/75 workstations, which are rated at 2 MIPS and have an Intel 82586 Ethernet controller. The ‘‘sun4’’ line is for SparcStation-1 workstations, which are rated at 12 MIPS and have an AMD Am7990 Lance Ethernet controller. The ‘‘ds3100’’ line is for DECstation 3100 workstations, which are rated at 13 MIPS and also have the Lance Ethernet controller.

when an additional ethernet packet has been used to transfer the data block. The maximum capacity of the 10 Mbit Ethernet is about 1200 Kbytes/sec. With 16 Kbyte blocks a Sun-3/75 attains 60% of this maximum, and the SparcStations attain 75%. Ordinarily Sprite transfers 4 Kbyte blocks because that is the file system’s block size. The latency and bandwidth for a 4 Kbyte block is 7.2 msec and 570 Kbytes/sec (4.5 Mbits/sec) on a Sun-3/75, and they are 5.37 msec and 745 Kbytes/sec (5.9 Mbits/sec) on a SparcStation.

3.6.2. RPC Traffic

The measurements presented in this sub-section are based on statistics taken from the system as it is used for day-to-day work by a variety of users. The raw data is in the form of counters that are maintained in the kernel and periodically sampled. File servers were sampled hourly, while the clients were sampled 6 times each day. The study period began July 8, 1989 and ended December 22, 1989.

The RPC traffic rate on all four file servers combined is shown in Figure 3-11. (Similar graphs for the individual servers are given in Appendix B.) Each line in the

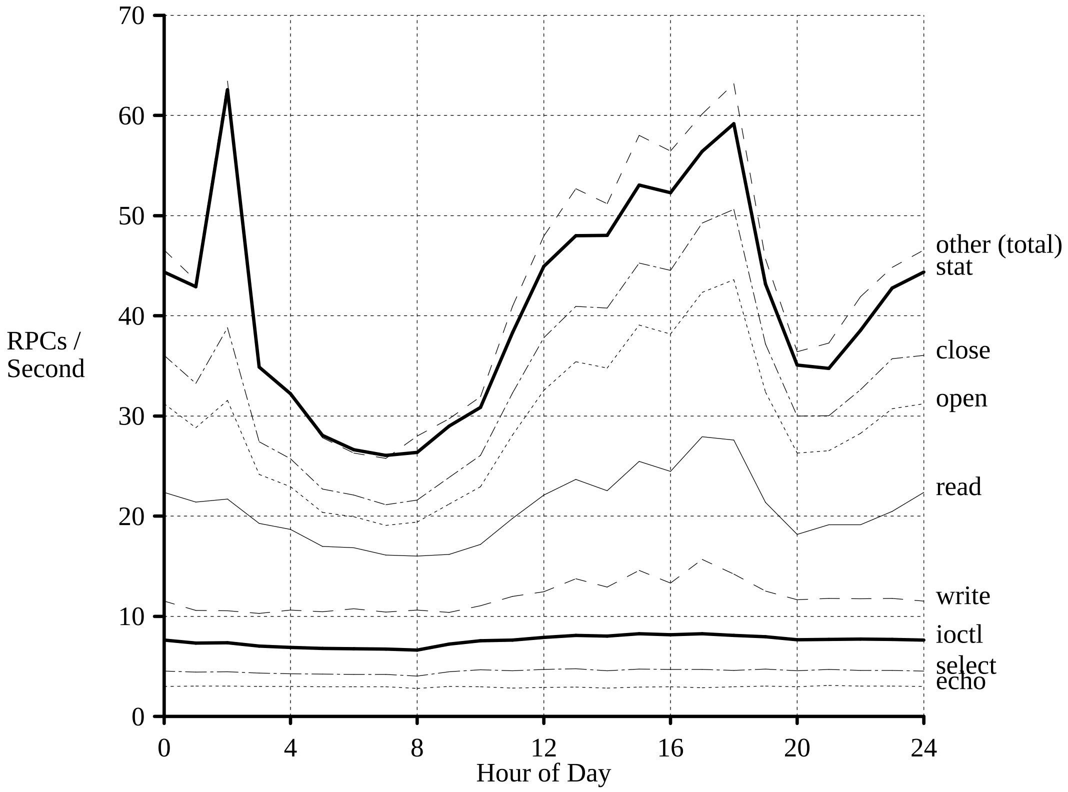


Figure 3-11. RPC traffic throughout the day for the combination of all four file servers. The servers were measured from July 8 through December 22, 1989. The lines for each RPC are cumulative, so the rate for each RPC is given by the difference between two lines. The top line represents the total RPC traffic.

graph is cumulative, so the difference between two adjacent lines gives the rate for the RPC that labels the upper line. The line labeled ‘‘other’’ also represents the total RPC traffic of the servers. The traffic represents sustained load as opposed to peak values because the statistics were sampled every hour and then averaged over months of activity. The traffic in the early morning hours gives a measure of background activity. The **echo** RPCs are generated by the host monitor module described in Chapter 6. The rest of the background stems from system daemons. Currently this load is rather high. Steps will need to be taken to reduce this load if we want to scale the system from its current size of 40-50 hosts up to our target of 500 hosts. For example, once a minute a daemon on each host updates a host status database, which is used by our process migration facility. Each update requires a number of file system operations in order to lock a record, read the old value, update the value, and unlock the record. **Ioctl** is used for locking. The nightly dumps are visible as the peak in **stat** (i.e., GET_ATTRIBUTES) RPCs. The reason there are more **open** than **close** RPCs is because of failed lookups and artifacts of the name resolution protocol, which will be described and measured in Chapter 4.

3.6.3. Other Components of the Architecture

This chapter has focused on the naming and I/O interfaces between the top-level file system routines and the object-specific modules, and the way that remote access is supported. There are also major modules in the architecture associated with the block cache, cache consistency, the disk sub-system, local name lookup, distributed name lookup, and pseudo-file-systems and pseudo-devices. The sizes of these modules are given in Table 3-2. Of these modules, the block cache, disk sub-system and local name lookup are relatively straight-forward implementations of concepts present in stand-alone UNIX. The block cache defines its own back-end interface so it can be used for both local and remote files, and perhaps it will be extended for other uses in the future (e.g., pseudo-file-systems). Most of the other file system components are unique to Sprite. The cache consistency module is described in detail in Nelson's thesis[Nelson88b]. Chapter 4 will describe the distributed naming module, and Chapter 7 will describe pseudo-file-systems and pseudo-devices. The ''utility'' module listed in the table includes routines that manage the table of object descriptors, and it includes the recovery protocol for the descriptors that is described in Chapter 6.

Size Comparison of FS Modules				
Module	Procedures		Lines	
Top level FS interface	82	13.8%	6301	14.3%
Object-specific (files pipes devices)	95	16.0%	5879	13.3%
Remote access	80	13.4%	5678	12.9%
Pseudo-file-systems and pseudo-devices	68	11.4%	5559	12.6%
Data block cache	62	10.4%	4672	10.6%
Disk management	53	8.9%	4557	10.3%
Local name lookup	27	4.5%	3577	8.1%
Utility routines	64	10.8%	3222	7.3%
Cache consistency	35	5.9%	2528	5.7%
Distributed naming (prefix tables)	29	4.9%	2078	4.7%
Total	595		44051	

Table 3-2. The number of procedures and lines of code in the modules that make up the file system implementation. About half of the lines are comments, and these are included in the line counts.

3.7. Comparison with Other Architectures

The Sprite file system architecture is similar to three UNIX-based architectures: the SunOS *vnode* architecture [Kleiman86]; the Ultrix *gnode* architecture [Rodriguez86]; and the AT&T *file system switch* (FSS) architecture [Rifkin86]. These are all generalizations of the original UNIX file system architecture. In each system, an internal interface was added below which file-system-specific dependencies are hidden. Different implementations of the internal interface allow for different file systems to be integrated together into one system as viewed by the user. There are two main differences between Sprite and these architectures. The first concerns remote device access, and the second concerns the way the distributed name space is implemented. Remote device access is discussed here, and the next chapter describes the Sprite naming mechanism and compares it with the UNIX-based approaches used by these other architectures.

The other architectures are oriented towards regular file access, and they make some assumptions that preclude the general remote device access provided in Sprite. The major assumption is that there is one server that assumes the role of the file server and (by assumption) the Sprite I/O server. Under this assumption it may be possible to access a remote device on the file server, but not on a different host (i.e., another workstation). Accessing a remote device on a file server is possible with the RFS file system protocol, which has been implemented in the FSS and vnode architectures[Chartock87]. However, there has to be distinct a ‘‘/dev’’ directory for each server so that the name for a device and the device itself are implemented by the same server.

The vnode and gnode architectures provide a way to access a local device that is named by a remote file server. This kind of access is needed to support diskless workstations. In this case, the equivalent of the Sprite IO_OPEN operation can ‘‘clone’’ a new descriptor from the descriptor returned from the file server, with the new descriptor being used for local device access. However, this still does not necessarily support the general three-party situation support by the Sprite architecture where a third host is the I/O server for the device.

Perhaps the stickiest issue related to remote device access concerns blocking I/O. Sprite provides a general blocking I/O mechanism that even applies to SELECT on devices on different hosts. In contrast, the other architectures allow the object-specific procedures to block indefinitely. To avoid using up all the server processes in RFS, for example, the last available server process is prevented from blocking by aborting an I/O request that would block. Also, SELECT is not supported in AT&T UNIX (which uses RFS), so the issue of having to wait on local and remote devices simultaneously is avoided. NFS, which is the remote file protocol used with the vnode and gnode architecture, does not support remote device access, so these architectures have also avoided this issue.

LOCUS is another UNIX-based distributed system. However, LOCUS does not define an internal interface that can be used to hide special-case processing. Instead, remote access was coded specially for each file system operation. One of the points in Walker’s thesis [Walker83b], for example, was how different remote operations in LOCUS were implemented with specialized message protocols. While SELECT is supported in LOCUS, there is no description of how they address the races associated with remote devices. Also, remote device access in LOCUS is coded within each device driver, while it is implemented by general-purpose code in Sprite. To be fair, the main focus of LOCUS was file replication and location transparency, not a software architecture. The LOCUS implementers modified an existing UNIX implementation as a fast route towards experimentation with file replication, and they did that before any of the UNIX-based architectures described above were introduced.

3.8. Summary and Conclusions

The architecture presented here is organized around two main internal interfaces, one for naming operations and one for I/O. These interfaces hide the special-case processing needed to support remote servers and different kinds of objects (i.e. files, devices, pipes, pseudo-devices). The use of distinct interfaces for naming and I/O enables the file servers to implement a name space that is used for non-file objects like devices. A file server can implement the naming interface for devices, while other hosts can implement the I/O operations for devices. The architecture supports our environment of diskless workstations; these hosts have devices like displays, printers, and even tape drives, but they do not have to implement any part of the name space so they can run diskless.

Remote access is implemented in a general way in the architecture, and this is hidden by the internal interfaces. Clients invoke RPC stub procedures through the interface, and on the server complementary RPC stubs call through the interface to invoke the service procedures. This approach means that the stubs can be shared by different object implementations (i.e., pipes and devices), and the service procedures are not littered with special case checks against the remote case.

The architecture supports state on the servers about I/O streams from remote clients. This state is used to support the caching system, it is used to implement exclusive access to devices, and it is used to prevent modification of files that are being used as program images. The state is maintained during Fs_Open and Fs_Close operations; there is no need to inform the server when I/O streams become shared due to stream inheritance. The way this state is recovered after server crashes is the topic of Chapter 6.

Remote blocking I/O operations are also supported by the architecture. A callback approach is used where I/O servers call back to a client that has been blocked. This frees up server processes while the client waits, and it supports Fs_Select operations on objects that are scattered around the network. The implementation of the callback has a potential race condition between a process’s decision to wait and a notification message from a remote server. This race is guarded against by remembering early notifications and surrounding blocking operations with a retry loop.

With this architecture as a framework, the remaining chapters consider some additional problems in more detail. These include name resolution in a system with many file servers (Chapter 4), the effects of process migration on the architecture (Chapter 5), and the failure recovery system for stateful servers (Chapter 6). Chapter 7 describes how user-level server processes can implement the internal naming and I/O interfaces in order to extend the basic functionality provided by the distributed file system with arbitrary, user-supplied services.

CHAPTER 4

Distributed Name Resolution: Prefix Tables

4.1. Introduction

This chapter considers the problem of name resolution in a distributed system composed of many servers, both Sprite file servers and pseudo-file-system servers. The goal of the naming mechanism described here is to hide the underlying distribution of the system. The result is that users and applications are presented with a single hierarchical name space that does not reflect its partitioning among servers, nor does the name space change when viewed from different hosts. All Sprite hosts share the same *location transparent* name space. Names do not reflect the location of an object, and the name of an object is the same throughout the network.

The hierarchical structure of the file system name space provides a natural way to partition it among servers. Each server implements a sub-tree of the hierarchy, and the sub-trees are combined into a global hierarchy. Sprite uses a system based on *prefix tables* and *remote links* to partition the name space [Welch86b]. Prefix tables on the clients provide a hint as to the server for the sub-tree identified by the prefix. Remote links are special files that indicate where other sub-trees are attached. Both the prefix tables and remote links are transparent to applications, so a single, seamless hierarchy is presented. The Sprite prefix mechanism has performance and administrative advantages over variants of the UNIX mount mechanism used in other systems, and it also differs substantially from the prefix mechanism used in the V-system.

The rest of this chapter is organized as follows. Section 4.2 describes the name resolution mechanism in stand-alone UNIX. Section 4.3 describes how this has been extended for use in a network by other systems, and points out drawbacks of this approach. Section 4.4 describes Sprite's prefix table mechanism. Section 4.5 presents measurements of the effectiveness of the system. Section 4.6 compares the Sprite prefix table mechanism with other prefix-based name resolution systems. Section 4.7 concludes the chapter.

4.2. Name Resolution in Stand-alone UNIX

The stand-alone UNIX directory searching algorithm is described here to provide a basis for the following discussion of distributed name resolution. There are three key points to the UNIX lookup algorithm.

(1) **Component-at-a-time-lookup.** A pathname is processed one component at a time in an iterative manner. The current directory in the pathname is scanned for the next component. The current directory is initialized to the root directory in the case of absolute pathnames (e.g., ''/a/b/c''), or to the process's working directory in the case of relative pathnames (e.g., ''a/b/c''). The lookup terminates when the final component is found in its directory, or access to a component of the path is denied.

(2) **Directory format**. Each directory contains a list of entries, with each entry consisting of a component and an identifier for a disk-resident descriptor. The identifier is a disk-relative index into the set of descriptors on the disk. When a component is found during lookup, its descriptor is fetched from disk so that its type and access permissions can be checked. This structure means that directories only reference objects on their own disk, so each disk has a self-contained hierarchical directory structure. Descriptors have a type such as file, directory, symbolic link, or device. Descriptors store other attributes of their object, including access permissions and the location of the object's data on disk.

(3) **Mount tables**. Hierarchical directory structures on different disks are combined into a single hierarchy with a mount mechanism. A main-memory mount table defines what directory structure contains the overall root, and it contains zero or more additional entries that define a connection between a leaf directory in one hierarchy and the root directory of another. The leaf directories named in the mount table are called *mount points*. When the lookup procedure encounters a mount point, the current directory is shifted to the root of the mounted directory structure before searching for the next component. The converse has to be done when traveling up the hierarchy due to parent directory (''..'') components. In this case the mount table is used to switch the current directory from the root of one hierarchy back to the directory it is mounted on before the current directory is advanced to the parent. For example, if ''/a/b'' is a mount point, then the lookup of ''/a/b/..'' will cross the mount point twice, once when processing component ''b,'' and a second time when processing the parent directory.

4.3. Remote Mounts

While the UNIX lookup mechanism was designed for a stand-alone system, it has been extended for use in a distributed file system. The mount table is extended to identify the server for a directory structure, and an additional mechanism is added for resolving pathname components in a remote directory structure. There are three different approaches to resolving remote components, which are described below.

The lowest level approach, which is used in LOCUS[Popek85], is to modify the routines that read directories to fetch them from the remote server instead of a local disk. Directories are cached in the main-memory file data cache, just as they are in UNIX, to avoid repeated remote operations[Sheltzer86]. LOCUS includes a cache consistency mechanism so that cached directories reflect additions and deletions made by other hosts.

The interface between client and server is low level. Server directories are directly manipulated by remote clients. The advantage of this interface is that the bulk of the directory lookup code from stand-alone UNIX remains unchanged. (LOCUS was implemented as a direct modification of UNIX.) The disadvantage is that all hosts must agree on the directory format, and modifying a directory incurs the cost of the cache consistency mechanism.

A slightly higher level approach, which is used in NFS[Sandberg85], is to send a component and a directory identifier to the remote server, which returns an identifier for the component if it finds the component within that directory. This interface between client and server is higher-level than the corresponding interface in LOCUS because the directory format is hidden from the clients. Servers scan their own directories, and they do their own directory modifications. The interface makes it easier to use NFS between different kinds of systems. A disadvantage of this approach, however, is that there can be many remote operations to resolve a pathname, one for each component. A cache of recent lookup results is kept by NFS clients so they can avoid some server operations. However, there is no cache consistency scheme in NFS so clients must refresh their lookup caches every few seconds, and it is still possible for them to use an out-of-date cache entry.

The highest level approach, which is used in RFS[Rifkin86], is to send the remaining pathname to the server when a remote mount point is reached. The RFS server processes as many components as it can and returns an identifier if it completes the lookup, or the remaining pathname if a ‘‘..’’ component caused the lookup to cross back over the mount point. The advantage of this approach is that it reduces the number of remote server operations. Only in diabolical cases where pathnames cross many mount points will there be many remote operations, and in these cases NFS would have at least as many remote operations (ignoring the NFS lookup cache). Another advantage is that there is no need for a directory consistency mechanism because the server does all its own modifications of the directory structure.

There is one overall disadvantage of the remote mount approach, however. Each host is free to define its own mount points so there is no guarantee that each host sees the same name space. This is a legacy of stand-alone UNIX where the mount table is initialized from a file on a local disk. In remote mount schemes, each host has its own mount table file and therefore each host defines its own name space. For example, host A could mount a directory structure under ‘‘/x’’, while host B could mount the same directory structure under ‘‘/r/s’’. This means that objects in the directory structure can have different names (e.g., ‘‘/x/a/b/c’’ vs. ‘‘/r/s/a/b/c’’) depending on which host is being used. In a network of workstations it is much better if each host sees the exact same name space so it does not matter which workstation is used. Achieving consistency of the name space with a remote mount scheme requires coordinated changes on all hosts in the network. Each host must update its mount table file and reinitialize its kernel data structures (perhaps by rebooting).

A common approach to this problem is to implement some sort of replicated database that defines the global mount configuration. A replicated database is used in LOCUS, the Andrew File System (AFS) [Satyanarayanan85], and others. However, replication is a heavy-weight approach that requires coordinated changes on many hosts. The Sprite prefix table mechanism, which is described next, takes a lighter-weight approach to name space consistency that is based on caching and lazy update.

4.4. Prefix Tables

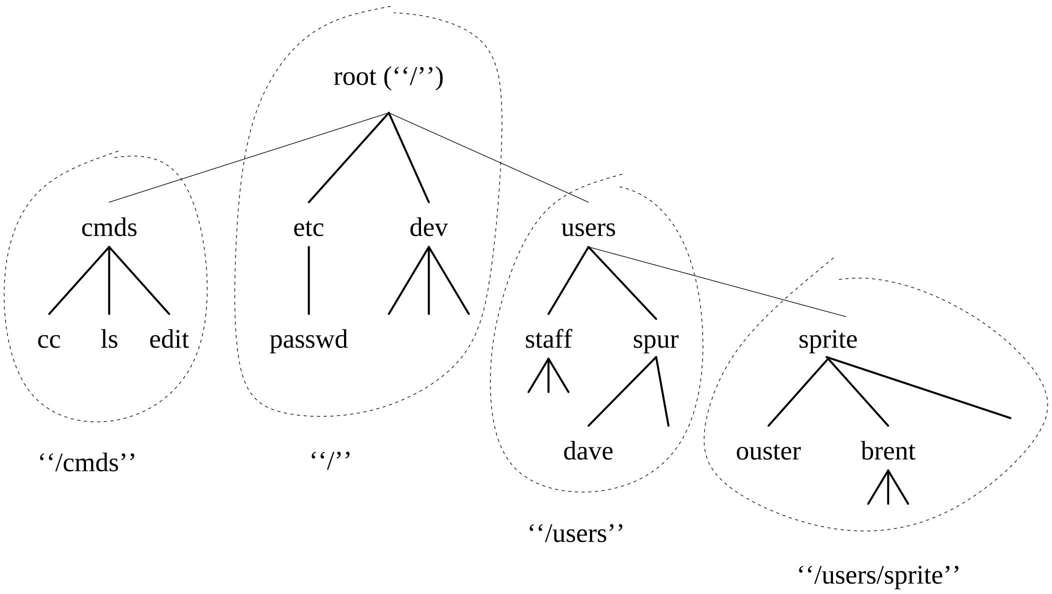
Sprite uses a *prefix table* mechanism to distribute the file system hierarchy over multiple servers. It is similar in function to the UNIX mount mechanism. It combines the directory structures on different disks, called *domains*, into a global hierarchy. With a mount mechanism, however, each host defines its own organization of different domains into an overall hierarchy, and lookup is done one component at a time while checking for mount points. With the Sprite prefix table mechanism, each domain is identified by a server-defined prefix that is the pathname to the root of the domain. Pathname lookup is divided into a prefix match, which is done by clients, and the resolution of a relative pathname, which is done by servers. Each host keeps a table of valid prefixes, and these tables are kept up-to-date using a broadcast-based protocol.

4.4.1. Using Prefix Tables

During file open and other operations that require name resolution, a client compares the pathname to its prefix table and selects the longest matching prefix. The longest prefix has to be chosen to allow for nesting of domains, as illustrated in Figure 4-1. For example, the prefixes ‘‘/’’, ‘‘/users’’, and ‘‘/users/sprite’’ all match on the pathname ‘‘/users/sprite/brent’’. The longest matching prefix, ‘‘/users/sprite’’, determines the correct domain. The client strips off the prefix and sends the remaining pathname to the server indicated by the matching prefix table entry. The server resolves as much of the pathname as possible. If it resolves the whole pathname the server performs the requested operation (e.g., NAME_OPEN, REMOVE, etc.).

A server may not be able to process the whole pathname, however, because a pathname can take an arbitrary path through the global hierarchy and cross through many different server domains. If a pathname leaves a server’s domain, the server returns the remaining pathname to the client. Returning a pathname is called *pathname redirection*, and the three cases for pathname redirection are described below in Section 4.4.3. Redirection results in an iterative lookup algorithm where the client applies the remaining pathname to the prefix table on each iteration in order to determine the next server to use. Control is returned to the client each iteration so that servers do not have to forward operations to other servers.

Note that a prefix match can cause the upper levels of the hierarchy to be bypassed during lookup. For example, our network has the server configuration given in Table 4-2. A lookup in domain ‘‘/sprite/src/kernel’’ is sent directly to server Allspice, and the three domains above it (‘‘/’’, ‘‘/sprite’’, and ‘‘/sprite/src’’) are bypassed. Bypassing directories reduces the load on the other servers. With remote mounts, in contrast, every component in an absolute pathname is processed so the upper levels of the directory



Prefix Table		
Prefix	ServerID	Token
‘‘/’’	A	<2>
‘‘/cmds’’	B	<753>
‘‘/users’’	C	<84>
‘‘/users/sprite’’	D	<17>

Figure 4-1. This figure shows the file system hierarchy and a prefix table that partitions the hierarchy into four domains. The distribution is transparent to applications. The server’s ID and a token that identifies the domain are kept in the prefix table.

Sprite Domains in Berkeley		
Server	Prefix	Kbytes
Mint	‘‘/’’	7820
Oregano	‘‘/a’’	264096
Oregano	‘‘/b’’	243936
Oregano	‘‘/c’’	300696
Allspice	‘‘/mic’’	625680
Assault	‘‘/dist’’	278520
Allspice	‘‘/swap1’’	275808
Allspice	‘‘/user1’’	305184
Assault	‘‘/user2’’	293832
Mint	‘‘/sprite’’	366160
Allspice	‘‘/sprite/src’’	627520
Allspice	‘‘/sprite/src/kernel’’	627520

Table 4-2. The Sprite domains in the Berkeley network. There are four file servers and 12 domains, which are shown here in order of increasing prefix length. Not shown are some smaller domains used for testing, and the pseudo-file-system domains that correspond to NFS file servers.

hierarchy are heavily used.

Bypassing the upper levels of the hierarchy with a prefix match also causes the access controls on the upper levels to be bypassed. All processes implicitly have search permission on the path to each domain. This lack of access control is not a problem in our environment, and it would require a per-process prefix table to close this loop-hole. Per-process access controls could be checked as a process's prefix table is initialized. However, a per-process prefix table would be less efficient than a kernel-resident prefix table. First, extra storage would be required because the prefix tables would not be shared. Second, extra processing would be required to initialize the prefix table for each new process. As described below, initialization involves a broadcast protocol so it is more efficient to share a prefix table among all processes and amortize the initialization cost over a larger number of naming operations.

The interface to the server is based on a relative pathname and a token. The token comes from the matching prefix table entry, and the relative pathname is obtained by stripping off the prefix. If the client already has a relative pathname, the token associated with the working directory is used. The tokens are descriptor IDs as defined in Chapter 3. They include a type, a server ID, and some server-defined information that identifies a directory. The token is used by the client to identify the server and its type. The type is used to branch through the internal naming interface to routines that access the server. The token is used by the server to determine at which directory to begin resolving the relative pathname. Prefix table initialization is described below.

4.4.2. Maintaining Prefix Tables

Clients initialize their prefix tables using a broadcast protocol. To create a new prefix table entry, a client broadcasts a prefix, and the server for the prefix, if any, responds with a token that corresponds to the prefix. The token is the descriptor ID for the root directory of the server's domain. The client saves the token in the prefix table entry and uses it as described above.

Prefix tables are maintained as a cache. Each entry is used only as a hint, which facilitates changing the server configuration. If the server for a prefix changes, out-of-date prefix table entries are updated by clients the next time they use them. In this case, a client gets an error (''invalid token'') when it uses the out-of-date prefix table entry, and it rebroadcasts to determine the new server. This lazy update approach eliminates the need for a complex distributed protocol to update the clients' prefix tables. Also, servers do not have to keep track of other servers, so there is no special inter-server protocol either. This technique applies to future lookup operations, but it does not help if the server for an open I/O stream changes. Chapter 6 describes how the file system's recovery protocol could be used to handle this problem.

A client learns about prefixes as a side-effect of pathname redirection. A client initializes its prefix table with ''/'', the prefix for the root of the hierarchy, and it broadcasts to locate the root server during system initialization. The root prefix matches on all pathnames so initially the client will direct all lookups to the root server. During pathname lookup, the root server will encounter a special file, called a *remote link*, at the point where another domain begins (in UNIX terminology, this is called a mount point). A remote link is similar to a UNIX symbolic link except that it contains a prefix and indicates that pathnames that cross the link are in another domain. The server combines the remaining pathname and the value in the remote link to create a new absolute pathname. It returns this expanded pathname to the client along with an indication of how much of the pathname is a valid prefix. The client adds the prefix to its table, broadcasts to locate the next server, and reiterates its lookup procedure. In this way, a client gradually adds entries to its prefix table as it accesses different parts of the file system hierarchy.

For example, suppose there is a domain identified by the prefix ''/users'', as shown in Figure 4-2, but the client only has ''/'' in its prefix table. The first time a client uses the pathname ''/users/sprite/brent'' it matches on ''/'', and ''users/sprite/brent'' is sent to the root server. The root server looks for the component ''users'' in its root directory and finds that it is a remote link with value ''/users''. The server expands the remote link to produce ''/users/sprite/brent''. This pathname is returned to the client along with an indication that ''/users'' is a valid prefix. The client adds ''/users'' to its prefix table, broadcasts to locate its server and get the token, and reiterates its lookup algorithm. On the next iteration ''/users/sprite/brent'' matches on ''/users'', and ''sprite/brent'' will be sent to the server for this other domain. Thus, the lookup iterates back and forth between the client and various servers as the pathname moves through different domains; there is no server-to-server forwarding of lookups.

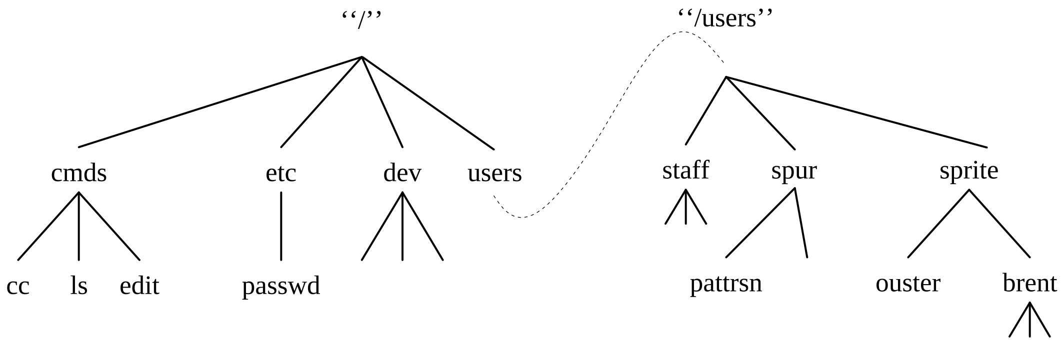


Figure 4-2. An example of a remote link. This file hierarchy is divided into two domains, ‘‘/’’ and ‘‘/users’’. The root directory contains a remote link that identifies the ‘‘/users’’ domain.

4.4.3. Domain Crossings

In the most general case a pathname can cross through many different domains before it terminates. There are three cases the system has to handle: 1) a pathname can descend into a domain from above, 2) a pathname can ascend out of a domain from below, or 3) a pathname can jump back to the root via a symbolic link to an absolute pathname. Descending into domains is detected when remote links are encountered, as described above. This case occurs if a client has not discovered the corresponding prefix, or if the lookup of a relative pathname begins in one domain and descends into another. In the latter case the server may return a prefix that the client already knows about, but this causes no problems.

Pathnames ascend out of a domain when they include ‘‘..’’, the parent directory. Each time the server processes a ‘‘..’’ component it checks to make sure it is not at the root of its domain. The way the server detects this situation is described below. If the server is at the root of its domain and the next component is ‘‘..’’, it returns the remaining name to the client. The client generates an absolute pathname using the domain’s prefix and the returned name. For example, if the server for domain ‘‘/a/b’’ returns ‘‘../x/y’’, the client combines these into ‘‘/a/b/../x/y’’, which further reduces to ‘‘/a/x/y’’. This pathname will no longer match on the prefix ‘‘/a/b’’, so the client will chose a different server for the next iteration.[11]

A symbolic link to an absolute pathname causes a jump to an arbitrary point in the hierarchy. The server expands the link and returns a new absolute pathname to the client.

[11] It would also be possible for the server to generate the new absolute pathname. That the client does this is a legacy of an earlier implementation where the server could not always tell what prefix a client had used.

This case is similar to the redirection that occurs when a remote link is encountered, except that no prefix information is returned to the client.

4.4.4. Optimizing Cross-Domain Symbolic Links

The measurements presented below will show that the bulk of pathname redirections in our environment stem from symbolic links between domains. The cross-domain links are artifacts of disk space limitations. Often several small directory structures are grouped together into one domain (i.e., one disk), and symbolic links are used to give the directory structures, or *sub-domains*, proper names. For example, in our network we have sub-domains corresponding to ‘‘/emacs’’ and ‘‘/X11R3’’, which store the source code and binary files for these software systems. We cannot afford to dedicate a whole domain to these directory structures, so we put them together into one domain (e.g., ‘‘/a’’). There are symbolic links from ‘‘/emacs’’ to ‘‘/a/emacs’’ and from ‘‘/X11R3’’ to ‘‘/a/X11R3’’. The unfortunate consequence of these symbolic links is that every lookup under ‘‘/emacs’’ or ‘‘/X11R3’’ will cause a pathname redirection. The lookup goes to the root server, which encounters the symbolic link and expands it (e.g., to ‘‘/a/emacs’’). The root server redirects the pathname back to the client, and the client issues the lookup to the correct server. Another significant example from our network is that ‘‘/tmp’’ was a cross-domain symbolic link to ‘‘/c/tmp’’. Consequently, all lookups of temporary files suffered a redirection.

It is possible to eliminate redirection in these cases by defining a new domain that corresponds to the target of the symbolic link. In our example, the server for ‘‘/c’’ can also export a sub-domain under the prefix ‘‘/tmp’’. The token associated with this prefix corresponds to the ‘‘/c/tmp’’ directory. The symbolic link from ‘‘/tmp’’ to ‘‘/c/tmp’’ is replaced with a remote link having the value ‘‘/tmp’’. The remote link causes clients to add ‘‘/tmp’’ to their prefix tables, and when they broadcast this prefix, the server returns the token corresponding to ‘‘/c/tmp’’. Future lookups will not require a broadcast or a redirection, and the server will resolve pathnames by starting at the ‘‘/c/tmp’’ subdirectory.

In effect, this approach caches the result of expanding the symbolic link in the client’s prefix table. It would be possible to cache the expansion of all absolute symbolic links like this. The server would expand the symbolic link and return it to the client as it does now, and the client would broadcast to get a token for the target of the symbolic link. The client would add the link’s pathname (e.g., ‘‘/tmp’’) to its prefix table along with the token for the target of the link (.i.e ‘‘/c/tmp’’). The main problem with this trick is that it bypasses the access controls on the path to the target of the link. A naive user could create a symbolic link into his directory structure and open it up to all other users. Because of this access control problem, domains are defined by system administrators by setting up the appropriate remote links and configuring the servers to respond to broadcasts for particular prefixes.

Exporting a subdirectory (‘‘/c/tmp’’) as a prefix (‘‘/tmp’’) makes it slightly more complicated for the server to detect when a ‘‘..’’ component is leaving a domain. It is not sufficient for the server to check against the primary root of a domain (‘‘/c’’); the root of the domain depends on what prefix the client used. In our example, when the user is in the ‘‘/c/tmp’’ directory, the parent directory is either ‘‘/c’’ or ‘‘/’’. The client has to identify two directories, the root directory for the domain and the starting directory for the pathname within the domain. Both directories are identified with tokens. If the client begins with an absolute pathname, the two tokens are the same. If the client begins with a relative pathname, one token corresponds to the working directory, and the root token corresponds to the prefix used when establishing the working directory itself. For example, when a process sets its working directory to ‘‘/c/tmp’’, the prefix that this name matched on (e.g., ‘‘/c’’) is remembered along with the token returned for ‘‘/c/tmp’’. On a subsequent lookup of a relative pathname the client sends the token for working directory ‘‘/c/tmp’’ to indicate where to start the lookup, and the token for the prefix ‘‘/c’’ to indicate the root.

4.4.5. Different Types of Servers

The location of the server (e.g., local or remote) is indicated by the type of the token in the prefix table. This type is used to invoke the correct service procedure in each iteration of the algorithm. The different cases are hidden from the top-level procedures that use the prefix table by the naming interface defined in Chapter 3. In the local case (i.e., on the file servers), the service procedures use a directory scanning algorithm much like the UNIX algorithm described in Section 4.2. The only modification is additional support for remote links and pathname redirection, and there is no use of a mount table. In the remote case a client uses RPC to invoke these service procedures on the file server. In a third case, described in Chapter 7, the naming operation is forwarded to a user-level server process.

4.4.6. The Sprite Lookup Algorithm

The features of the naming mechanism are summarized by the following lookup algorithm. The inputs to the algorithm are a pathname and per-process state information that indicates the working directory and user authentication information used for access control. The results of the algorithm depend on the particular naming operation. Operations like REMOVE and MKDIR change the directory structure and are implemented as the last part of resolving the name. In these cases the result is simply an error code. Other operations like NAME_OPEN and GET_ATTRIBUTES return information about the named object. These operations have been described in detail in Chapter 3. The resolution algorithm proceeds as follows.

(1) If a pathname is absolute, the longest matching prefix determines the server. The prefix is stripped off and the token associated with the prefix is obtained.

(2) If the pathname is relative, it is unaltered and the token that identifies the working directory is obtained.

(3) The server is given the relative name and the token that identifies the starting directory of the relative name. The server follows the pathname through the directory structure on its disks. If the pathname terminates in the server’s

domain, it performs the requested naming operation (e.g., NAME_OPEN, REMOVE, MKDIR, etc.).

(4) If the pathname leaves the server's domain, the server returns a new pathname and possibly a prefix to the client in a pathname redirection.

(5) The client updates its prefix table, if needed. If the prefix is new to the client, it broadcasts to locate the prefix's server and obtain the token for the domain. The client restarts the lookup at step 1 with a new absolute pathname. The new pathname will match on a different prefix so the lookup will eventually complete. (Symbolic links can create circularities in the directory structure. The client guards against these by limiting the number of redirections that can occur in a single lookup.)

4.4.7. Operations on Two Pathnames

A more complicated iteration over the prefix table may be required with operations that involve two pathnames (i.e., HARD_LINK and RENAME), but in the best case these can be done with a single server operation. The idea is that both pathnames (and their starting directory tokens) are sent to the same server, and in the best case the server can complete the operation. However, either or both of the pathnames might leave the server's domain and cause pathname redirection. In this case the client must reiterate over its prefix table as described below.

The HARD_LINK and RENAME operations require their two pathnames to lie in the same domain. HARD_LINK creates a new directory entry that references an existing object. The identifier in a directory is domain-relative, so it is impossible to create a ‘‘hard’’ link to another domain. (Symbolic links reference an object by name so there is no restriction on the target of a symbolic link.) RENAME is used to change the name of an object without copying it. RENAME makes a link to the object under the new name, and then deletes the original directory entry for the object. Thus, both of these operations fail if the two pathnames are not in the same domain.

The algorithm used for HARD_LINK and RENAME proceeds as follows. It has to handle redirections involving both pathnames, plus it has to detect when the two pathnames terminate in different domains.

(1) The prefix table is used to determine the server for the two pathnames. Both pathnames, and their associated prefix tokens, are sent to the server of the first pathname. In the best case both pathnames are entirely within this server's domain and the operation completes.

(2) If the first pathname leaves the server's domain, the server returns a new name to the client and does no processing of the second pathname. The client then restarts at step 1, updating its prefix table if needed.

(3) If the first pathname terminates in the server's domain, but the second pathname leaves the server's domain (or it does not even begin in the server's domain), the server returns a ‘‘cross domain’’ error.

(4) At this point the client has to verify the error. It is still possible that the second pathname can end up back in the same domain as the first pathname. This condition is tested by looking up the *parent directory* of the second pathname using a regular GET_ATTRIBUTES operation. The pathname of the parent is computed simply by trimming off the last component of the second pathname. The parent is checked in order to avoid ambiguous errors that can arise when checking the second pathname directly. With HARD_LINK, for example, the second pathname typically does not exist, so a lookup would return ‘‘file-not-found.’’ This may or may not mean the operation can complete, whereas a ‘‘file-not-found’’ on the parent directory definitely means the operation cannot succeed.

(5) If the parent directory of the second pathname is in the same domain as the first pathname then the client restarts at step 1. By this time the iterations between various servers have expanded both pathnames fully and established enough prefix table entries to direct both pathnames to the same server. Otherwise, if the second pathname is in a different domain, the operation cannot complete and the ‘‘cross domain’’ error is returned to the user.

An alternative approach to these two operations would be to build them up from low-level operations. However, this approach requires more than one server operation, even in the best case. It also requires that the server keep some state between low-level operations. This makes it more difficult for the server to implement HARD_LINK and RENAME atomically; it has to handle error cases where the client gets part way through the operation and then crashes.

4.4.8. Pros and Cons of the Prefix Table System

The prefix table system described here has four main advantages.

(1) **Simple Clients**. The client side of the lookup algorithm is simple, and diskless clients are easily supported. A client only has to be able to match prefixes, reiterate a lookup after pathname redirection, and broadcast to locate servers. The details of the directory format and symbolic link expansion are hidden from clients.

(2) **Name Space Consistency**. Servers export a domain under the same prefix to all clients so all clients have the same view of the file system name space. It does not matter what workstation a user uses. File servers share the name space so system administration chores can be done on any host. Servers do not have to worry about consistency of directories when they are updated because directories are private to the server.

(3) **Dynamic Reconfiguration**. It is easy to change the configuration of the system because clients’ prefix tables are refreshed automatically. Adding a new domain requires the addition of a remote link and updating the server’s set of exported domains. No system-wide reconfiguration procedure is needed.

(4) **Efficient Lookups**. Most lookups are performed with a single server operation. The prefix match bypasses the upper levels of the hierarchy, reducing traffic to

the root server.

There are three main disadvantages to this system:

(1) **No Name Caching**. Clients contact a server for every lookup operation. There are cases where clients open the same files many times in a short period of time, and caching lookup results would help reduce server traffic. However, a lookup cache adds complexity to both the client (which now has to use the cache) and the server (which has to ensure that client caches remain consistent). The data cache consistency algorithm also depends on seeing every **open** and **close**, and it would need to be modified to account for name lookup caching. While we considered adding a name cache, we avoided it because of the additional complexity it would add to the system.

(2) **Broadcast**. The use of broadcast limits a Sprite file system to a local area network capable of broadcast.[12] Mechanisms used in widely distributed name services could be employed to address this limitation [Oppen83][Lampson86]. However, there are more issues than the mechanisms of server location that have to be addressed for large scale distribution of the file system. In particular, system administration will require cooperation among the various groups using the system. Security and access will be more important issues as the user community gets larger. Basically, there is a whole step up in complexity associated with very large scale distribution that is not addressed in this dissertation.

(3) **Availability**. Failure of a server can prevent a client from getting new prefixes so some name lookups may not work until the server is restarted. (This recovery is handled automatically, as described in Chapter 6.) Availability of the system could be improved by replicating important domains, such as the root domain or the domain containing system programs. However, replication requires some mechanism to update the replicated domains, and some interserver negotiation to decide who will service what clients. Replication would add complexity to the implementation, and possible overhead to ordinary operations, so it has been avoided. However, as Sprite is scaled to larger networks in the future, the issue of replication may have to be addressed.

4.5. Measurements

A number of statistics are maintained by the hosts in our Sprite network, which shed insight on the effectiveness of the prefix table system. These statistics were sampled

---

[12] Currently, Sprite does support sharing of domains between hosts not reachable by broadcast, which is necessary because a few Sprite machines are in a remote laboratory. However, our current solution is rather crude and it ought to be replaced. We have added the capability to predefine the server for a prefix. In this case, the server is contacted directly the first time the prefix entry is used, and broadcast is not used. This feature was the quickest and simplest way to support distant machines, but it suffers from the maintenance problems of a remote mount system.

every hour (day and night) on the file servers and 6 times during the day on the clients (every three hours from 8:00 AM to 10:00 PM). The samples were taken from July 8 through December 22, 1989. The configuration of the file servers during this time is given in Table 4-3. The most heavily used server averages over 1 million RPC requests each day, so the data presented here are averages over many millions of operations. More detailed tables of the results presented in this Chapter are presented in Appendix B, Section 3.

File Server Configuration				
Name	Model	Memory	Disk Capacity	File Types
Mint	Sun-3/180	16 Mbytes	300 Mbytes	Root and commands.
Oregano	Sun-3/140	16 Mbytes	900 Mbytes	/tmp and source code.
Allspice	Sun-4/280	128 Mbytes	2400 Mbytes	User files and source code.
Assault	DECStation 3100	24 Mbytes	600 Mbytes	User files.

Table 4-3. The configuration of file servers during the study period. Mint has the root directory and most commands and system-related files. Allspice has most of the source code for the system, user files, and swap directories. Oregano has /tmp and some sources. Assault has user files.

4.5.1. Server Lookup Statistics

The average length of the pathnames processed by servers is between 2 and 3 components per pathname. Looking up a whole pathname with one RPC saves as much as 2 to 3 times the RPC traffic over a lookup system that does lookup one component at a time, depending on the effectiveness of a component name cache. Also, these average lengths do not count the components that matched on the client's prefix table, so there is even more savings over a component-at-a-time lookup.

The lookup traffic for the file servers is presented in Figure 4-3. The rates show a peak during the nightly dumps, and a hump during the daytime hours. A per-month and per-server breakdown of these statistics is given in Appendix B.

Almost 20% of all pathnames are not found. This percentage does not count the number of times a file was newly created. These failed lookups are an artifact of the directory search paths used by our compilers and command interpreters. A search path is a list of directories in which to look for commonly used files like command programs and compiler include files. Search paths let users specify files and commands with relative names (e.g., ‘‘ls’’ vs. ‘‘/sprite/cmds/ls’’). However, search paths also cause many failed lookups.

There is a significant rate of pathname redirections. About 13.5% of all lookups are redirected due to absolute symbolic links. Only 0.04% of all lookups are redirected due to remote links, and only 0.15% of all lookups are redirected due to leaving a domain via the parent of its root. Thus, the remote links are effective in loading the clients' prefix tables, and it is rare that a domain is entered or left via a relative name. The redirections due to absolute symbolic links are mainly due to a number of auxiliary commands

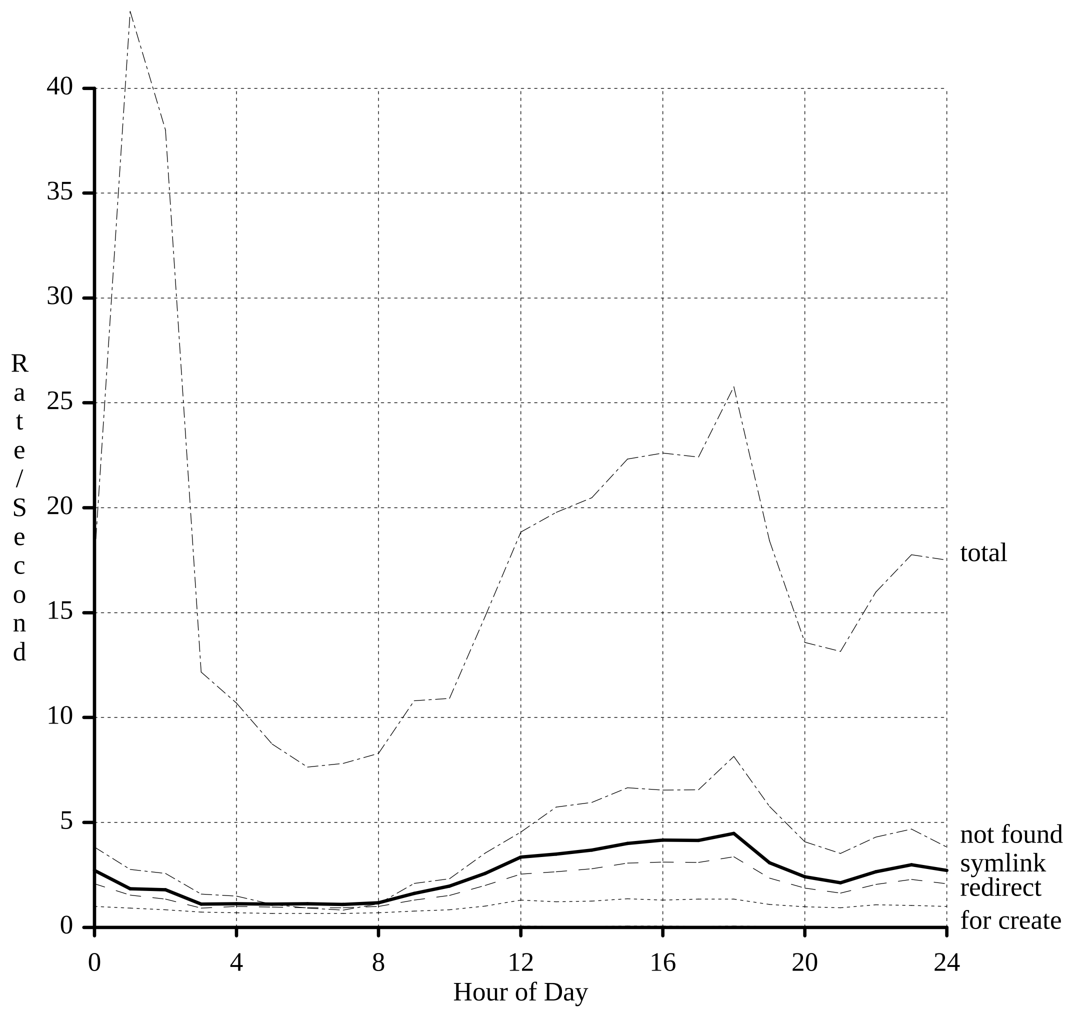


Figure 4-3. Lookup rates for the combination of all four file servers. The servers were measured from July 8 through December 22, 1989. Rates are shown for the total lookups, failed lookups (file-not-found), symbolic links traversed, pathname redirections, and open-for-create.

directories (e.g., ‘‘/emacs’’) that are accessed via symbolic links. We should be able to reduce the number of redirections caused by these symbolic links by exporting their target directory under a prefix, as described above in Section 4.4.4.

4.5.2. Client Lookup Statistics

Clients recorded the number of absolute and relative pathnames that passed through the naming interface, and the number of lookup requests that were redirected. These

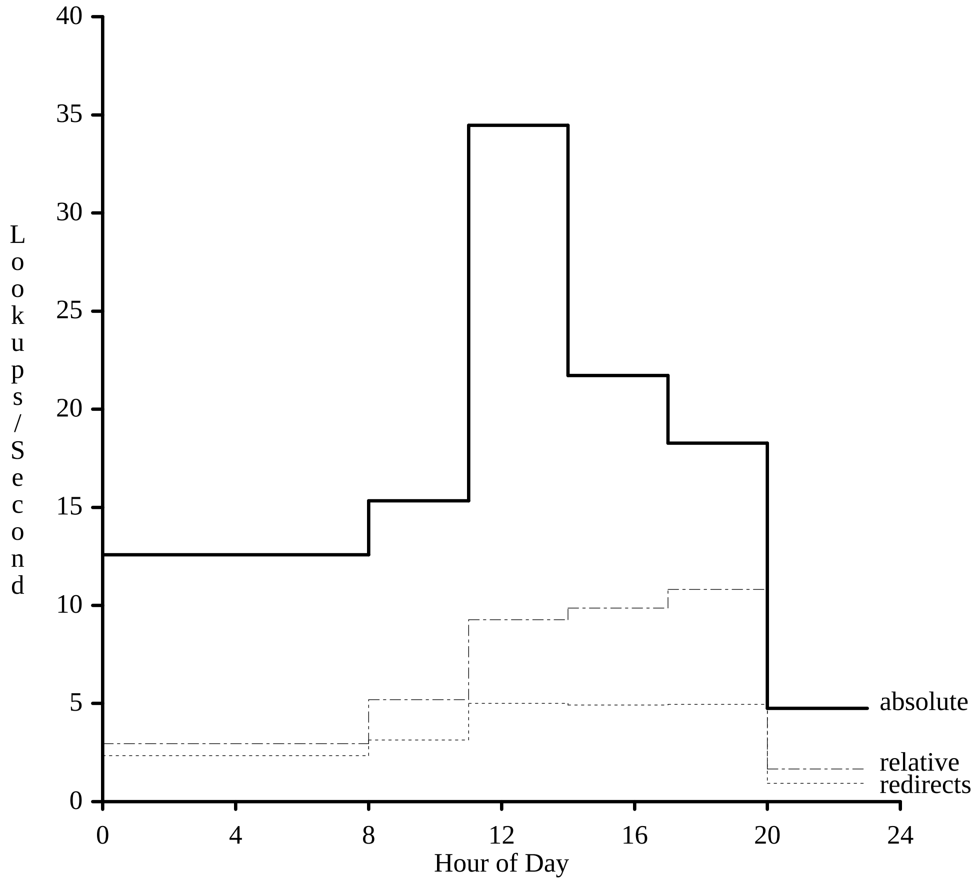


Figure 4-4. Lookup rates for 35 clients combined. The lookups are divided into those on absolute and relative pathnames. The rate of pathname redirections is also shown. These rates were computed from July 8 through Dec 22, 1989. Clients were sampled every three hours from 8:00 AM to 11:00 PM.

results are given in Figure 4-4. A per-client breakdown of lookup rates is given in Appendix B. Overall, 71% of the pathnames looked up by clients were absolute. This is due to command execution and traffic to system files like sources and libraries. If we assume that most pathnames are outside the root domain (the small size of the root domain precludes keeping much there), then the root directory is usually bypassed by the prefix match or the use of a relative name. Also, given the relatively short pathnames processed by the server, the elimination of even a single component by the prefix match significantly cuts the processing done by the servers.

4.6. Related Work

This section contrasts the Sprite prefix table mechanism with other distributed name lookup systems.

4.6.1. Remote Mounts

The differences between Sprite prefix tables and remote mount schemes should be apparent from the above descriptions of the two mechanisms. The primary advantage of Sprite prefix tables over remote mount schemes is that they ensure a consistent name space because the server defines the prefixes. In contrast, in a remote mount scheme the client decides where a remote file system is mounted into its own name space. An extra mechanism is required to ensure consistency of mount tables, such as the network configuration protocol used in LOCUS[Popek85].

4.6.2. Prefix Based Systems

The V-system has a general naming protocol that uses prefix tables[Cheriton89]. Prefixes in V are used to identify different types of servers (e.g., print servers, file servers, device servers). A system-wide name service is used to register server processes under prefixes. Prefixes are explicit in V-system names (e.g., ‘‘service]rest-of-name’’, where ‘]’ is a reserved character), so the mechanism cannot be used to transparently distribute a file system hierarchy. Instead, a client’s prefix table is used as a cache of results obtained from the name service.

The Andrew[Satyanarayanan85] and Echo[Hisgen89] file systems use prefix tables to distribute a file system hierarchy as in Sprite. However, these systems use a replicated name service to store the global configuration of the system so they can distribute their file systems over a wide area network. Ubik [Kazar89], which is used in Andrew, is a special purpose database that only allows one update at a time. This assumes that the file server configuration changes infrequently. The name service used by Echo[Lampson86] allows concurrent updates, but it provides *eventual* consistency of the name space. Updates are propagated as soon as possible, and the system guarantees that all name servers will eventually have the same information. This approach assumes that users will tolerate brief inconsistencies in the view of distant parts of the system.

The Sprite prefix table mechanism differs from V in that it provides transparent distribution, and it differs from Andrew and Echo because it is optimized towards the local area network environment. The Sprite naming protocol is unique in its use of remote links to trigger pathname redirection and define domain prefixes; it does not require the complexity of a replicated database.

4.7. Summary and Conclusion

This chapter has described a name resolution system based on prefix tables and remote links. There are a number attractive features of the system.

- It supports simple clients (i.e., diskless workstations).
- It ensures a consistent view of the name space on all hosts.
- It is easy to manage because prefix tables are automatically updated when the configuration of the system changes.
- It reduces the load on the servers for the upper levels of the hierarchy because the prefix match bypasses them.

The measurements from our system indicate that remote links are effective in loading the prefix tables. In only 0.04% of the server lookups were remote links encountered. Similarly, leaving a domain via its root is also rare; it happened in 0.15% of the lookups. However, 17.03% of the pathnames traversed a symbolic link, and 13.52% of the pathnames were redirected because of a cross-domain link. The large amount of redirections from symbolic links suggests that we can further optimize our domain structure by exporting important subdirectories like ‘‘/emacs’’ and ‘‘/X11R3’’ as sub-domains instead of accessing them via symbolic links.

To conclude, the prefix table mechanism is a simple mechanism that is used to transparently distribute the file system’s hierarchical name space across the network. The mechanism is really a simple name service that is closely integrated with the regular name lookup mechanisms used in traditional, stand-alone file systems. The prefix table is implemented as a thin layer on top of the routines that manage a directory structure on the local disk. However, the clean split into the client and server parts of the system make it possible to merge the directory structures maintained on different servers into a single hierarchy shared by all the clients.

CHAPTER 5

Process Migration and the File System

5.1. Introduction

Process migration is the movement of actively executing processes between hosts [Douglis87]. We use migration daily in our system to move compute- and file-intensive jobs from our personal workstations to other workstations that are idle. A migrated process can continue to use I/O streams to file system objects and open new I/O streams using the same names it would have used before it migrated. There is no need to copy files between hosts or to restrict access to devices because of the remote access facilities of the file system and the shared file system name space. Thus, the combination of process migration and the shared file system allow us to exploit the processing power available in our network.

When a process migrates, it is necessary to migrate its open I/O streams, which is a complex procedure. The precise state that the file servers keep to support caching and crash recovery has to be maintained during migration. Updating this state requires coordinated actions at the I/O server and the two clients involved in migration. In addition, the procedure has to be serialized properly with state changes due to other operations. The locking done during migration has to mesh correctly with the locking done by other operations, otherwise deadlock can result. This chapter describes the deadlock-free algorithm by which I/O streams are transferred to new hosts.

Migration can cause streams to be shared by processes on different hosts, and additional state is required to support the semantics of the shared stream in this case. The current access position of the stream is shared by the processes sharing a stream. In the absence of migration, the shared offset is supported by keeping the access position in the stream descriptor and sharing the stream descriptor among processes. Migration can cause processes on different hosts to need access to the shared stream access position. This chapter describes a mechanism based on *shadow stream descriptors* that is used to maintain the shared access position. A client's stream descriptor is bypassed and the server's shadow stream descriptor is used when the stream is shared by processes on different hosts.

The remainder of this chapter is organized as follows. Section 5.2 explains the problem of stream sharing in more detail and describes the use of shadow stream descriptors to solve it. Section 5.3 describes a deadlock-free algorithm to update the file system's distributed state during migration. Section 5.3 discusses the cost of migration to the file system, and Section 5.4 concludes the chapter.

5.2. The Shared Stream Problem

If two processes share a stream, I/O operations by either process update the stream's current access position. Normally the access position is kept in the stream descriptor, and all processes sharing the stream reference this same descriptor. Now, suppose one process migrates to another host. One reference to the stream descriptor migrates to a new host, while other references remain on the original host. Some mechanism must be added to the system to support the shared stream access position among processes on different hosts.

One approach, which is used in LOCUS, is to circulate a *stream token* among the different hosts sharing the stream[Popek85].[13] The token represents ownership of the stream, and it has to be present on a client before an application can use the stream. If the token is not present then the I/O server is contacted, and it retrieves the token from the current owner. In this case, there is an additional overhead of 2 RPCs every time the stream changes ownership. Also, if a client is writing to the stream, any data it has written has to be flushed back to the I/O server when it gives up the token. LOCUS uses a token circulation scheme for cache consistency, as well. Associated with each file is one write token and one or more read tokens, and the LOCUS I/O server is in charge of fairly circulating the stream token, read tokens, and write token among clients.

The other approach, which is used in Sprite, is to keep the stream access position at the I/O server in a *shadow stream descriptor* when the stream is shared. In this case, I/O operations go through to the server because it keeps the current stream access position. This approach requires an RPC on each I/O, but there is no need for callbacks from the I/O server to other clients. The shadow stream solution parallels the technique that Sprite uses for consistent data caching. The problem in data caching is to keep the cached data consistent with other copies cached elsewhere. To simplify data cache consistency, the local data cache is bypassed if a file is being write-shared by processes on different hosts. Similarly, the local stream descriptor can be thought of as a cache for the stream's access position. When a stream is shared by processes on different hosts, the local stream descriptor has to be bypassed to get the most up-to-date stream access position. These two cases are treated identically by the implementation, so the effects of stream sharing on the data caching mechanism are very small in terms of additional code complexity. Thus, Sprite bypasses the local cache during sharing, while LOCUS circulates a token to represent ownership, and it always accesses the local cache.

The amount of message traffic required with the Sprite scheme will be less if the stream is heavily shared, even if the stream is read-only. In Sprite, each I/O would require one RPC. In LOCUS, each I/O would require 2 RPCs to retrieve the token (one from the client to the server and one from the server to the other client that holds the token), and perhaps an additional RPC to fetch the data if it is not in the cache. This comparison assumes the worst case for LOCUS where processes on different hosts

[13] The term ‘‘offset’’ token is used in [Popek85], not ‘‘stream’’ token. Offset is another term for the current access position of the stream.

interleave their use of the I/O stream. On the other hand, if the stream is really only used by one of the processes that shares it, the LOCUS scheme may have less message traffic. Once that process obtains ownership of the stream, it can continue to do I/O without RPCs, assuming that all the data for the stream is in the local cache. Finally, if the I/O stream is to a device and not a file, then Sprite's scheme will always require less message traffic. The I/O operations always have to go through to the device's I/O server, so there is no benefit from caching the stream access position on the clients.

In our network virtually all the cross-machine stream sharing occurs within a single application, **pmake**. **Pmake** uses migration to dispatch compilations and other jobs to idle hosts. **Pmake** generates a small command script, forks a shell process that will read the script, and migrates the shell to an idle host. The I/O stream to the command script is shared by **pmake** and the shell. The script is generated before the shell is migrated, so it is written to the cache of **pmake**'s host, and then written back to the file server during the migration. The way this happens is explained in more detail below. The read operations by the shell, however, have to go through to the file server because of the shared stream.

Measurements from our network indicate that very little I/O is done to streams shared by processes on different hosts. In a one-month study period this I/O traffic accounted for only 0.01% of the bytes read by clients. This low traffic suggests that a simple mechanism to share the stream offset may be the best, since the mechanism is unlikely to affect overall system performance. The Sprite scheme is simpler because the I/O server does not have to make callbacks to retrieve tokens, nor does it have to worry about circulating the token fairly among the processes that demand the token.

5.2.1. Shadow Stream Implementation

In the following discussion, the term ‘‘stream reference’’ is distinguished from ‘‘stream descriptor.’’ It is the stream references that migrate among hosts, not the stream descriptor, and the system must support stream references on different hosts that reference the same stream descriptor.

The addition of shadow streams to the kernel data structures is shown in Figure 5-1. Note that the shadow stream always exists on the I/O server, whether or not migration has occurred. This is a simplification in the implementation, and its effects are discussed below. The shadow stream has a client list that is used to detect when the stream is in use on different client hosts. The following invariants, which must be maintained during migration and error recovery, apply to the shadow stream descriptor.

- The stream access position is valid in the client's stream descriptor if the stream is only in use at that client. Otherwise, the stream access position is valid in the server's shadow stream descriptor.
- The shadow stream has a client list with an entry for each host with a process that is using the stream. If a stream has two references and one reference migrates to another host, the client list on the shadow stream descriptor must be updated to reflect this.

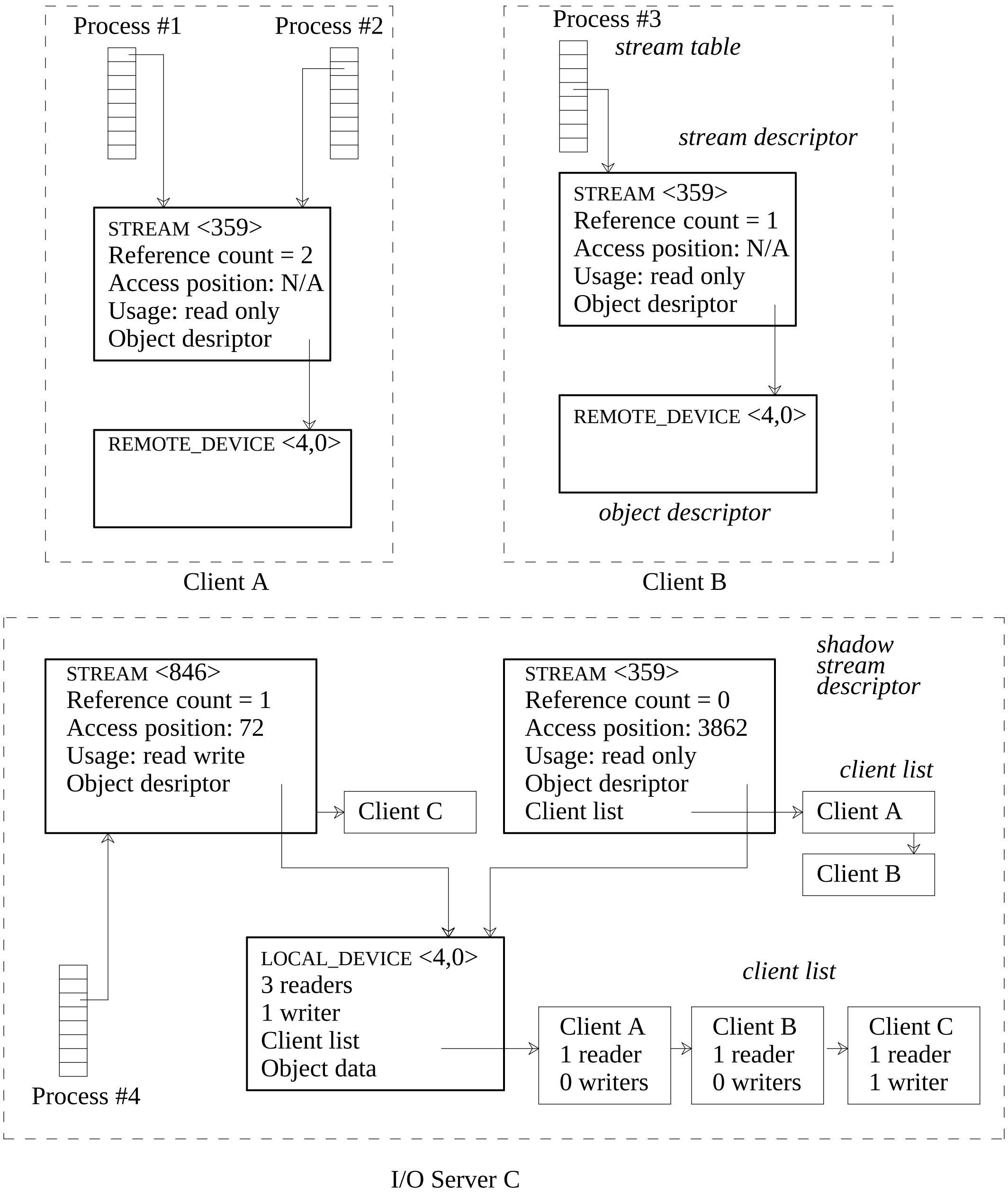


Figure 5-1. Streams and shadow streams. This is similar to Figure 3-6, except that stream #359 is now shared by processes on different hosts, and there is an additional shadow stream descriptor for this stream on the I/O server. The shadow stream descriptor has its own client list that is used to detect when the stream becomes shared by processes on different hosts.

- The reference count on a shadow stream descriptor does not reflect remote processes that are using the stream. Only local processes, if any, are reflected in the shadow stream's reference count. This invariant ensures that remote FORK and DUP operations do not have to be propagated to the I/O server. Only during migration (and OPEN and CLOSE) is there communication with the I/O server.

Currently a shadow stream descriptor is created on the I/O server for every stream, not just those that are shared during migration. The main reason the shadow descriptor is always created is to simplify the RPC interface to the server. All I/O requests identify both the stream and the underlying object, and the server verifies both the existence of its shadow stream and the corresponding object descriptor. However, it increases the memory usage of the servers to always have shadow stream descriptors. In the future, the implementation may be optimized to create shadow stream descriptors on demand, but the following description assumes they are always created.

5.3. Migrating an I/O Stream

This section describes the procedure used to migrate a stream reference to a new client. The main problem is that the I/O server treats references to the same stream on different clients as different streams. Regardless of what clients share a stream's access position, the I/O server has to account for stream references on different clients to properly support cache consistency. For example, a file that is being both read and written is still cacheable on a client as long as all the readers and writers are on the same client. However, as soon as a single stream reference migrates to a different host the file is no longer cacheable. In effect, migration can cause a stream to split into two streams as viewed by the I/O server. Such a split is shown in Figure 5-2. Migration can also cause two streams to merge back into one, in the case where all the stream references end up back on the same client.

Figure 5-2 is a simple example of a stream with 2 references that demonstrates the problem. Before migration the stream descriptor has a reference count of 2, and there is 1 usage count on the object descriptor from the stream. If both references are closed, the first CLOSE will simply decrement the reference count in the stream descriptor and have no effect on the object descriptor. The second CLOSE will take away the last stream reference, at which point the usage counts on the object descriptor have to be decremented. However, if one of the stream references migrates to a new host, then each host will have a stream descriptor with a reference count of one, and closing either stream will result in a CLOSE being made on the object descriptor. Thus, what used to be viewed as a single stream at the level of the object descriptor is now two distinct streams. In order to support this the client list on the object descriptor has to have an additional entry added for the new client, and the summary usage counts on the object descriptor have to be incremented to account for the additional CLOSE that will be seen.

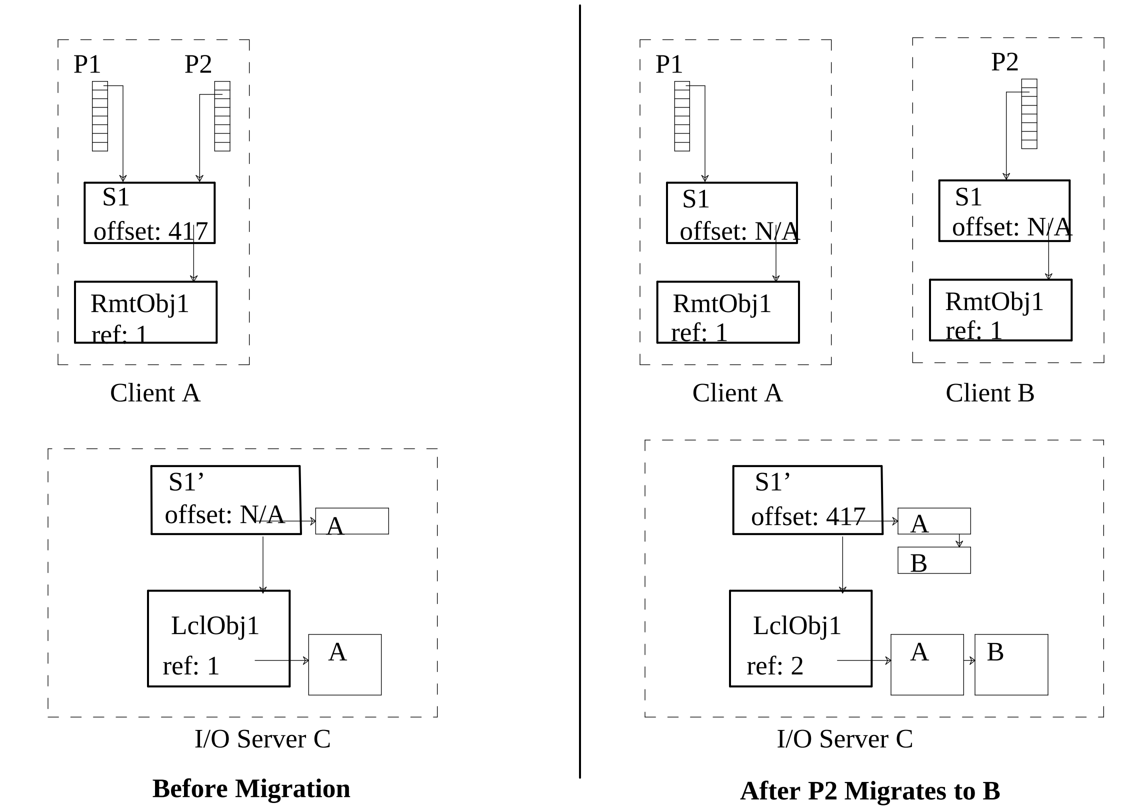


Figure 5-2. A stream may be shared by processes on different hosts after migration. Before the migration the I/O server only knows about the stream on one client, and the fact that the stream is shared by two processes is hidden from it. After migration the stream is in use on different clients and the I/O server has to be informed because it is as if a new stream were being opened at the second client. The I/O server adds a new entry to its client list and increments is summary usage counts to reflect the ‘‘new’’ stream.

5.3.1. Coordinating Migrations at the I/O Server

The basic approach to updating the state of the I/O data structures during migration is as follows. When a process migrates away from the source client no changes are made to the source client’s I/O data structures, nor is the I/O server contacted. Instead, the destination client notifies the I/O server of the migration after the process arrives there. The I/O server then makes a callback to the source client. At this point coordinated state changes occur. During the callback the I/O server holds its shadow stream descriptor locked in order to serialize with migrations and closes of different stream references. The result of the callback indicates if any stream references remain at the source client. The I/O server updates the client list in the shadow stream descriptor to reflect the new stream on the destination client, and possibly the departure of the stream from the source client. After the shadow stream client list is updated the I/O server can determine if the stream is in use on different clients. If it is, then it must update the client list on the underlying object descriptor and take any object-specific actions that are required. In particular, the cache consistency algorithm is invoked at this time in the case of regular files.

One of the important points of the above algorithm is that the state of the system is updated after a process arrives at its destination, and no state changes are made as it is leaving the source client. Unrelated state changes can occur between the time a stream reference leaves the source client and arrives at the destination. Other stream references could be closed at the source, or they could migrate onto the source client, or I/O operations could update the stream access position. The I/O server has to coordinate these various cases, and coordination is most easily done after a reference has arrived at the destination.

The stream access position is also shifted among hosts during migration. When the first stream reference migrates away from the source client, the server fetches the current access position as part of the callback. The access position remains valid at the server until all the stream references have migrated to the same destination client. The server passes the access position to the client in the return parameters of the destination client's notification. This callback structure is illustrated in Figure 5-3.

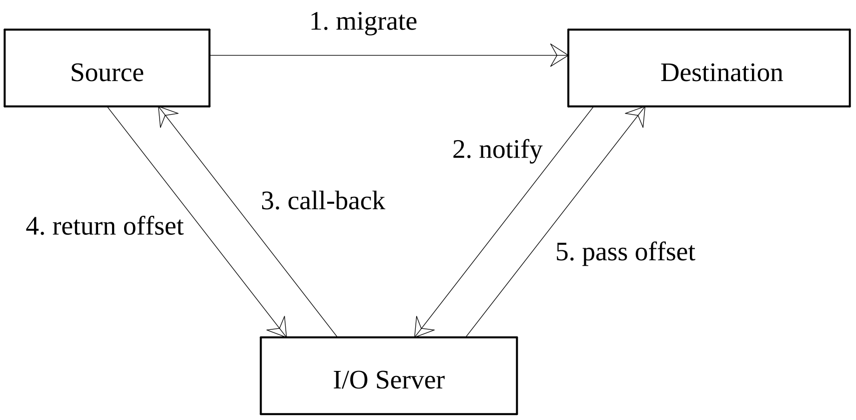


Figure 5-3. Migrating the stream access position. The destination client notifies the I/O server that a stream reference has migrated to it. The I/O server makes a callback to the source client to retrieve the stream access position and tell the source to release a stream reference. If the only stream references are at the destination client, then the I/O server passes the offset to the destination client.

5.3.2. Synchronization During Migration

Proper synchronization during migration is critical. The file system is implemented using locks on individual data structures to allow for concurrency on a multiprocessor. There is a conflict between holding many locks in order to ensure that state changes on the clients and the I/O servers are consistent, and preventing deadlock with other operations that can be proceeding concurrently. A potential source of deadlock is the interaction of a CLOSE by the source client with the callback by the I/O server during a MIGRATE. During CLOSE the client locks its stream descriptor, and if it is the last stream reference the client also locks the underlying object descriptor, then the I/O server locks its shadow stream descriptor, and finally the I/O server locks its object descriptor. These locks are held so the object descriptors can be updated to reflect the departure of the stream from the source client. During the callback the I/O server locks its shadow stream descriptor and then the client attempts to lock the its stream descriptor and the its object descriptor. The shadow stream is locked during the callback to serialize this migration with other migrations and closes of other stream references. There is apparent circularity in lock dependencies here, but the following two scenarios show that deadlock cannot occur.

The first case concerns the CLOSE and MIGRATE of references to the same stream. Deadlock cannot occur because the CLOSE only goes through to the I/O server when the last reference is being removed from the stream descriptor. If there were two references at the source client, for example, and one was migrating at about the same time as the other was being closed, then either of the following two situations occurs. If the CLOSE occurs before the callback from the server, then it will see a reference count of 2 at the source client, and it will not contact the I/O server. When the CLOSE completes, the callback can lock the client's stream descriptor, discover that the last reference has migrated away, and destroy the stream descriptor. If the callback occurs first, it will decrement the reference count at the source client and unlock the stream there. The CLOSE will then go through to the I/O server and be blocked on the locked stream descriptor there. However, the callback has been made, so when the MIGRATE completes, the CLOSE operation will proceed.

The second case concerns the CLOSE and MIGRATE of different streams to the same object. Deadlock cannot occur because the I/O server leaves its object descriptor unlocked during the MIGRATE callback. The CLOSE of a different stream will lock the client's object descriptor and then attempt to lock the I/O server's object descriptor. If the I/O server locked its object descriptor and then attempted to lock the client's object descriptor during the MIGRATE callback, then deadlock could result. Thus, it is crucial that the I/O server's object descriptor is not locked during the MIGRATE callback so that a CLOSE of an unrelated stream will not hang the callback.

5.4. The Cost of Process Migration

There are a few important costs to the file system that stem from process migration. Additional complexity is required to maintain the shadow stream descriptors and the stream accounting during migration. Additional storage space is required on the I/O servers to store the shadow stream descriptors. Finally, there is some overhead on server operations because the client must be validated as a proper client of the shadow stream.

5.4.1. Additional Complexity

The additional complexity due to migration is difficult to measure accurately. One measure is in the additional code required to support it. Migration accounts for about 6% of the code (2617 lines) and 7% of the procedures (41) in the file system implementation. About half of the migration code is object-specific, and these procedures often re-use existing object-specific routines. The bulk of the complexity is in general routines that handle the synchronization problems, the shared access position, etc.

Size is not necessarily an accurate measure of complexity, however. Migration was a difficult piece of the file system to implement correctly because it interacted with many other areas of the file system. The bookkeeping that supports caching had to be properly maintained, and the locking of data structures to serialize operations had to done correctly during migration, too. The initial implementations contained deadlocks, which stemmed from a conservative approach to locking data structures. In a highly concurrent environment, such as that on the file server, it is tempting to hold locks on multiple data structures to be sure that state changes are properly coordinated. Furthermore, because migration changes state on different machines, locks are held on multiple machines simultaneously. There were a number of attempts at the migration algorithm that seemingly worked, only to deadlock days later after many thousands of successful migrations.

5.4.2. Storage Overhead

The shadow stream descriptors can occupy a significant amount of memory on the file servers for two reasons. First, file servers always create shadow stream descriptors, whether or not migration will occur, in order to simplify the implementation. However, the file servers can have several thousand shadow stream descriptors during periods of heavy activity. A Sprite client workstation with an average window system setup has around 200 streams. About 70 of these streams are purely local and are associated with the window system and TCP/IP. However, there are around 130 streams that have shadow descriptors on a file server. The number of streams can increase considerably during large jobs, and our file servers have peaked at nearly 8000 shadow stream descriptors.

The second reason the stream descriptors occupy a lot of space is because they are rather large, about 100 bytes each. The size of the stream descriptor is impacted by two implementation issues, both of which could be optimized in the case of shadow stream descriptors. First, stream descriptors are implemented as a standard object descriptor so they can have a descriptor ID and be manipulated with the standard utility routines to add them to a hash table, fetch them, lock and unlock them. The header information for a standard descriptor is 40 bytes, including a 16 byte descriptor ID. Secondly, there is some naming information associated with a stream that is only needed in certain circumstances, and it is not needed on a shadow stream descriptor. This is another 40 bytes or so. All that is really needed on a shadow stream descriptor is an ID, a reference count, some flags, a client list header (head and tail pointers), and a pointer to the object descriptor. The ID could be implemented as an <I/O server, number> pair to ensure uniqueness, so the shadow stream descriptor could be trimmed down to 28 bytes assuming 4-byte integers and pointers.

The storage associated with the shadow stream descriptors is a pragmatic issue, but one that has to be addressed as the system scales up to support a larger network. In our current network of around 30 clients, peak values of 8000 streams indicate that a system with 500 clients could have well over 100,000 streams requiring 10 Megabytes of memory for the shadow stream descriptors alone. At this scale there is more motivation to trim down the storage associated with stream descriptors. The best optimization would be to delay creation of the shadow stream until migration occurs.

5.4.3. Processing Overhead

The shadow stream descriptor also adds overhead when servicing clients' RPC requests. Clients identify both the stream and the underlying object in RPC requests, and the server is careful to verify that the client is on the client list of both the shadow stream descriptor and the object descriptor. This checking is done to ensure that the client's stream data structures match the servers because a diabolical failure could cause the client and server to become out-of-sync. The server rejects the client's request if the server's state does not agree with the client's, which forces the client to take the recovery actions described in Chapter 6 to resolve any inconsistencies. The server also has to check if the offset in its shadow stream descriptor should override the offset provided by the client. These actions add a small amount of overhead on each remote I/O operation, whether or not migration is being used.

5.5. Conclusions

This chapter described the use of shadow stream descriptors to solve the problem of preserving the semantics of shared I/O streams during process migration. The shadow stream mechanism parallels the solution to the shared file caching problem. If a stream is shared among processes on different hosts, the stream descriptors on those hosts are bypassed and the server's shadow stream descriptor is used. Similarly, if a file is write-shared among processes on different hosts then their caches are bypassed and the server's cache is used. This technique of bypassing the local cache during sharing is a simplification; it eliminates the need for circulating ownership of the current access position (or file data) among clients.

This chapter also presented a deadlock-free algorithm for updating the file system's distributed data structures during migration. The procedure is complicated by concurrent operations such as closes and other migrations that also affect the file system's distributed state. The I/O server coordinates updates to the data structures on both clients involved in migration, and it locks its own data structures carefully to avoid deadlock while still providing the needed synchronization with concurrent operations.

CHAPTER 6

Recovering Distributed File System State

6.1. Introduction

This chapter considers the effects that host failures have on the distributed file system. There may be many clients and servers in the system, and it is not uncommon for machines to be rebooted for routine reasons, or for outright failures to occur. This chapter describes a new recovery system that relies on redundant state in the network and cooperation between clients and servers. Clients can continue some operations when servers are down, and they recover automatically after servers restart. For example, users can continue to use interactive applications such as editors while a server is down, perhaps for a routine reboot. Most of the processes already running on their workstation, including the window system and the editor, can continue execution. A background compilation, however, might be blocked because it needs to access a file on the server. It will be continued automatically after the system recovers. This chapter describes the mechanisms built into the file system that provide this kind of fault tolerance.

The recovery problem is complicated by the ‘‘stateful’’ nature of the Sprite file servers. The servers keep state about I/O streams to their files and about how clients are caching file data [Nelson88a]. A server’s state has to be maintainted across crashes and reboots. Otherwise, open I/O streams to the server would be destroyed by the server’s loss of state. The RFS [Back87] and the MASSCOMP remote file systems [Atlas86] handle server failures in this way. This behavior is not acceptable in our environment of diskless workstations. The crash of an important server could wipe out most of the active I/O streams in the system and essentially require a restart of all workstations.

A standard approach to preserving the server’s state across crashes is to log it to stable storage. Logging is used in MFS, Burrows’ stateful file system[Burrows88]. However, logging slows down the OPEN and CLOSE operations because of the additional disk I/O’s required. Instead, this Chapter describes a different approach to recovery that relies on redundant state on the clients instead of logging:

> Server state is organized on a per-client basis, and each client keeps a duplicate version of the server’s state about the client. This state can be kept in main-memory on the server and the client in order to avoid the costs associated with disk accesses. The server can clean up the state associated with a client if the client crashes, and the client can help the server rebuild its state if the server crashes.

The state that is protected in this way in the Sprite file system are the I/O stream data structures, file lock information, and dirty cache blocks. The client’s object descriptor is augmented with enough information to rebuild the state kept in the client list of the server's descriptor. The dirty cache blocks are retained on the client until the server has written them safely to disk. (Actually, a Sprite file server implements a number of writing policies that trade off performance for reliability. This is discused below in Section 6.4.) Keeping the extra state on the client adds little processing overhead during normal operations and requires no additional disk operations.

There are two additional problems related to error recovery that are considered in this chapter. The first is detecting the failure of other hosts, which is done by monitoring the RPC communication protocol. The system recognizes two failure events, crashes and reboots. Client crashes trigger clean-up actions by servers, while server reboots trigger recovery actions by clients. Measurements presented below indicate that there is very little overhead from failure detection.

The second problem concerns operation retry after server recovery. Operations that access a failed server are blocked and then retried automatically after server recovery. However, users are given the option of aborting operations instead of waiting for server recovery. The recovery wait is structured in the same way that blocking I/O is structured, at a high level, outside object-specific code and with no resources locked. The recovery wait can be aborted easily with this structure. This approach provides flexibility in handling errors so that users are not necessarily stuck when they attempt to access a failed server.

The remainder of this chapter is organized as follows. Section 6.2 contrasts the stateless and stateful server models. Section 6.3 describes the state recovery protocol. Section 6.4 discusses the interactions of delayed-write caching and error recovery. Section 6.5 discusses operation retry after recovery completes. Section 6.6 describes the crash detection system that triggers state recovery. Section 6.7 describes our experiences with the recovery system. Section 6.8 concludes the chapter.

6.2. Stateful vs. Stateless

The difficulty of recovering state after failures has lead to the notion of a ''stateless'' server, one that can service a request with no state (or history) of previous requests.[14] What this means in practice is that the server logs all essential state to disk so the state will not be lost in crashes. The advantage of statelessness is that failure recovery is merely a matter of restarting the server, and clients do not have to take any special recovery actions. WFS [Swinehart79] and NFS [Sandberg85] are examples of stateless file servers. In these systems, a client can simply retry an operation until it gets a response from the server. The operation retry is handled inside the communication

[14] Neither ''stateful'' or ''stateless'' are in my dictionary. Even if they were defined with their assumed meanings, with state and without state, they would not be completely accurate; all servers keep state of some sort. However, ''stateful'' implies that some essential state may be lost in a server crash.

protocol, and a server failure is manifested only as an especially long service call. Eventually, the server reboots itself and, by its stateless nature, can service any client request (e.g., READ or WRITE). In contrast, if a stateful Sprite file server crashes, it cannot service a READ or a WRITE until it has reestablished some state about the client, the state that was established during an OPEN. An additional recovery protocol is required to return the Sprite server's state to the point before it crashed.

While a stateless server makes recovery easier, it suffers a performance hit because it has to write all important state changes to disk. The performance disadvantage is demonstrated by comparing the NFS and Sprite file systems. Both systems use main-memory caches to optimize file I/O, but the stateless model of NFS limits the effectiveness of its caches. NFS uses write-through caching, while Sprite uses delayed write (or write-back) caching. When an NFS file is closed, the client must wait for all its dirty blocks to be written to the server's disk. The server can forget about the WRITE and remain stateless. With Sprite's delayed write approach data is allowed to age in the cache without being immediately written through to the server. Results from Chapter 8 indicate that about half the data is never written out to the servers. Instead, it is deleted or overwritten before it ages out of the cache. Head-to-head comparisons with NFS in [Nelson88a] show that Sprite is 30%-40% faster than NFS, mainly because of these different writing policies. New results from faster machines (10 MIP as opposed to 2 MIP) indicate that Sprite may be as much as 50-100% faster in the remote case on the new fast workstations available today[Ousterhout89b]. Network and disk speeds are not increasing as fast as CPU speeds. Sprite's caching mechanism, which eliminates network and disk accesses, tends to scale performance with CPU speed, while the NFS protocol is limited by disk and, to a lesser degree, network speeds.

Hardware such as RAM disks with battery backed-up memory can be used to improve the performance of a stateless file server. The state can be kept in the non-volatile memory, which is treated as a very fast disk. Addition of special hardware, however, is an expensive solution, and it does not address the issue of maintaining perfect consistency of the client caches, which is the goal of the Sprite file servers' state.

6.3. State Recovery Protocol

The following sub-sections describe how the system's state is organized to allow recovery and the steps taken during the recovery protocol. Recovery actions are triggered by events from the kernel's failure detection system, which will be described in detail in Section 6.5. The failure detection system generates two events, crash events and reboot events. Crash events trigger clean-up actions by servers, while reboot events trigger recovery actions by clients.

6.3.1. Recovery Based on Redundant State

Sprite's approach to recovering state is to duplicate the server's state about a client on that client, and then rebuild the server's state from the copies on its clients. To support this approach, the client and the server maintain parallel state information: the servers keep per-client stream information, and the clients keep state about their own streams.

There is very little added to the I/O stream data structures described in Chapter 3 to support this approach. The server's state is already organized into client lists to support the cache consistency algorithm. All that is required is for the client to duplicate the accounting information kept in the server's client list entry. What would otherwise be a simple reference count on the client's REMOTE object descriptor has to be a full fledged set of usage counters, just like the ones the server keeps. In the discussion below the usage is simplified to be a reader count and a writer count, but the system also records the number of streams from active code segments, and information about user-level file locks. The addition to the client's REMOTE descriptors can be stated as another invariant on the file system state. An example is given in Figure 6-1.

- The usage counts in a client's REMOTE object descriptor summarize the number of streams that reference the object descriptor. The usage count does not reflect stream sharing among processes. For example, even though stream #359 is shared by two processes, it only counts as one stream in terms of object use. The client's usage counts are used to recover the server's client list.

The stream access position is also vulnerable to a server crash while the stream is shared by processes on different hosts. In this case, the stream access position is cached in the server's shadow stream descriptor, and the access position in the client's stream descriptor is not used. However, the clients have to keep backup copies of the stream access position if the shadow stream descriptor is to be fully recoverable. The server has to return the current stream access position to the client after each I/O operation so that if the server crashes, the access position is not lost. A server-generated time stamp also has to be returned so the server can determine which access position is most up-to-date upon recovery.[15]

6.3.2. Server State Recovery

There are two parts to maintaining the servers' state during crashes. The first concerns cleaning it up after a client crashes, and the second concerns repairing a server's state after it reboots or a network partition ends. Cleaning up after a dead client is relatively straight-forward; in response to a crash event the server examines its client lists and closes any open I/O streams and releases any file locks that were held by the crashed client. This cleanup is implemented by calling an object-specific CLIENT_KILL procedure on each object descriptor so the cleanup actions can be tailored to the specific needs of files, devices, and pseudo-devices. The CLIENT_KILL procedure is an addition

[15] This aspect of recovery, however, is not currently implemented. If the server crashes when a stream is shared across machines, then it will loose the shared access position. Most shared streams are for a process's current working directory, which does not depend on the access position. However, this bug could cause errors and it ought to be fixed.

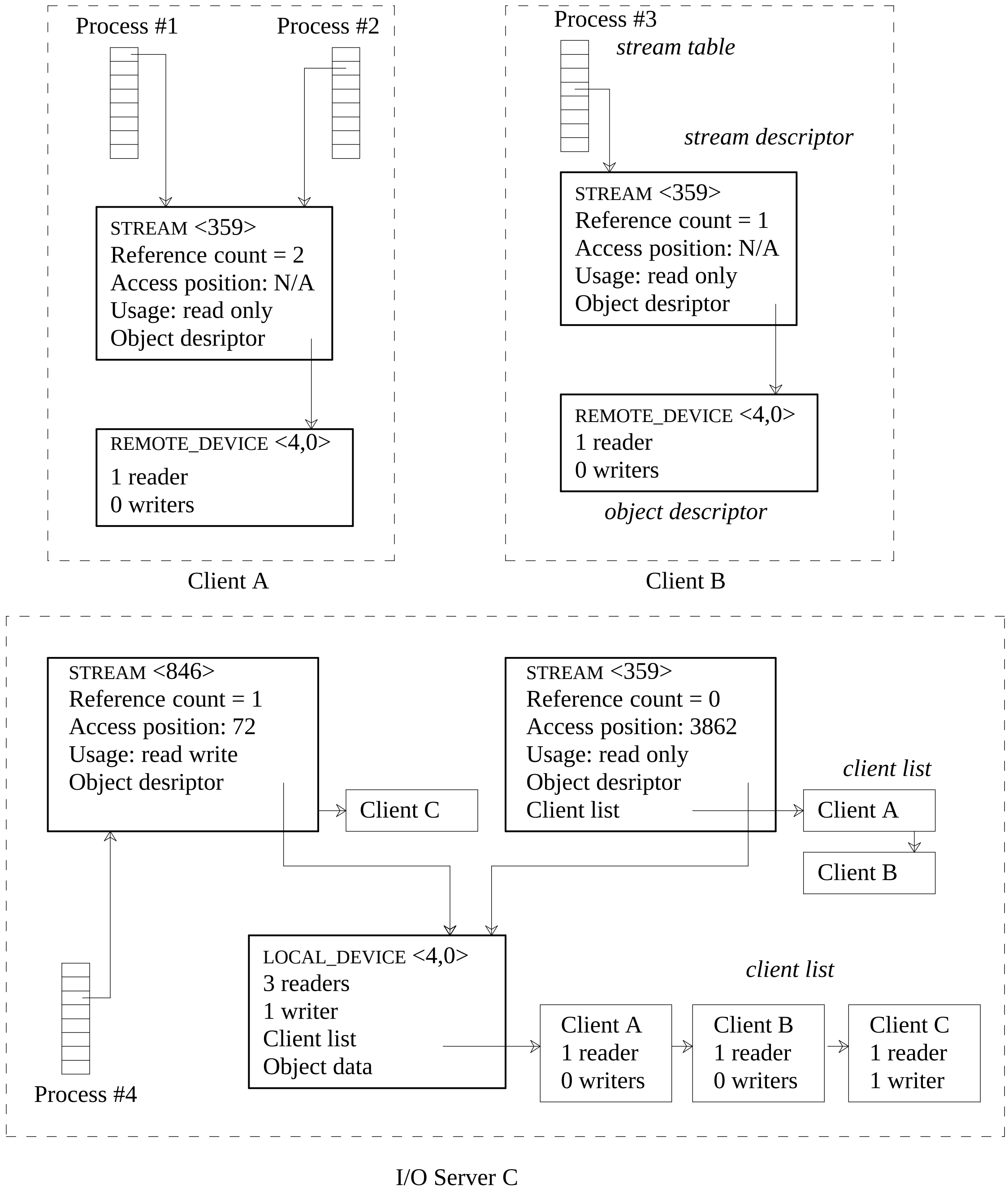


Figure 6-1. REMOTE object descriptors on clients are augmented to keep usage counts that parallel the usage counts kept in the client list of the server's LOCAL object descriptor.

to the I/O interface described in Chapter 3.

Recovery of a server's state is done with an *idempotent* recovery protocol. An idempotent operation can be invoked many times and have the same effect as invoking it one time. In this case, idempotency means that the recovery protocol always tries to reconcile the state on a client with the existing server state. Idempotency is important because network failures can (and will!) lead to unexpected inconsistencies between the client's and the server's state. In particular, the operations in progress at the moment of a network failure may or may not complete on the server, so the client cannot be sure of the server's state after the network partition ends. An idempotent recovery protocol means that the clients can be conservative and invoke the recovery protocol whenever they think the server's state might not be consistent with their own.

Clients initiate recovery in response to a reboot event or when they get a ''stale descriptor'' error from a server. The ''stale descriptor'' error indicates that the server does not have the client registered on the client list of its object descriptor. This situation can arise if the server has invoked the CLIENT_KILL procedure in response to a crash event generated during a network partition. (As described below in Section 5.5, the failure detection system cannot distinquish between host crashes and network partitions.)

6.3.2.1. The Initial Approach

In the current implementation, clients perform recovery by invoking an an object-specific REOPEN procedure on each of the client's object descriptors that were opened from the server that rebooted. REOPEN is an addition to the I/O interface described in Chapter 3. The object-specific implementation of the REOPEN procedure is similar for all cases (i.e., files, devices, pseudo-devices); the summary use counts in the client's REMOTE object descriptor are sent to the server via RPC, along with any object-specific information required by the I/O server. At the I/O server a complementary REOPEN procedure is invoked to determine if the client's use of the object is ok. The server's REOPEN procedure compares the client's usage counts with the ones it has in its client list. The server may have to close streams, open streams, or do nothing at all in order to bring its object descriptor up-to-date with the client's. Opening and closing streams involve object-specific actions. The actions associated with cached files are described below, in Section 6.4.2.1.

The server's state also includes the shadow stream descriptors that were introduced in Chapter 5 to keep a shared stream's current access position. Currently the client makes a pass through its stream descriptors after it has reopened all of the underlying object descriptors. During this phase the server verifies that it has a corresponding shadow stream descriptor.[16] Errors during this phase cause the client's stream to be closed and the server's client list to be adjusted accordingly. Errors can arise if the server

---

[16] It is also during this phase that the server should recover the shared access position in the shadow stream.

accidentally reuses a streamID for a different shadow stream. While this case is rare, the current implementation is conservative and it checks for this conflict anyway.

Note that this two-phase approach (object descriptor recovery then stream descriptor recovery) requires the client to block OPEN and CLOSES during the recovery protocol. This ensures that the two phases of its recovery protocol give the server the same view of the client's state. As described below, this restriction could be eliminated.

The main problem with this recovery protocol is that it places more load on the servers during the recovery protocol than necessary. There are two contributions to the extra load. First, clients can have a large number of unused object descriptors because of the data cache. The descriptors contain state about the cache blocks of a file, and the descriptors are cached along with the data. These descriptors are reopened as part of the recovery protocol, even though it would be possible to revalidate them the next time they are opened. Second, there is some redundency in this recovery protocol. First, the client reopens the object descriptors, which contain counts of I/O streams that reference these descriptors. Second, the client reopens its stream descriptors to ensure that the server has corresponding shadow stream descriptors. Thus, the server hears about a stream twice, once when the object descriptor is reopened and once when the shadow stream descriptor is reopened. The result of these inefficiecies is that our servers often see tens of thousands of REOPEN requests in a short period of time after they reboot.

6.3.2.2. Improvements

The existence of shadow streams suggests an alternative structure to the recovery protocol, although this alternative has not been implemented. The client could perform one REOPEN for each of its stream descriptors first and eliminate the need for a REOPEN of each of its object descriptors. The stream REOPEN operation would identify the object descriptor that it references and indicate the usage of the stream. Idempotency would be achieved by checking for the existence of a shadow stream descriptor on the server. If one exists then the server already has state about the stream and it would ignore the REOPEN. Otherwise the server would open a new stream and update its client list accordingly. This approach eliminates both problems described above; cached but unused object descriptors would not be reopened because no stream would reference them, and servers would only hear about each stream once. Additionally, this approach eliminates the need to block out OPEN and CLOSE operations during the recovery protocol.

The main drawback of this approach is that it depends on the existence of the shadow stream descriptors. As noted in Chapter 5, there can be a large number of shadow stream descriptors if there is one for each client stream instead of just having one for each client stream that is shared due to migration. However, Sprite already relies on large main memories for its data cache [Nelson88a] so it is not unreasonable to assume there will be enough memory for the server's shadow stream descriptors.

6.3.2.3. Reopening Cached Files

When reopening regular files the client's RemoteFileReopen procedure passes to the file server the summary use counts of the file (i.e., how many readers and writers), lock information, the version number of the client's cached copy, and a flag that indicates whether or not the client has dirty data cached. At the file server, the LocalFileReopen procedure is invoked in response to the client's RPC. This procedure invokes the regular cache consistency protocol to ensure that the client sees the most up-to-date version of the file, and it updates the file's client list to agree with the client's reader and writer count, unless a conflict is detected.

The cache consistency protocol detects conflicts that can arise after a network partition. If a client caches a file for writing and a network partition occurs, dirty data can be trapped in the client's cache. During the partition the server may get a crash event from the host monitor and close the I/O streams from the client. This action allows another client to open the file for writing and generate a conflicting version of the file. A difference between the recovering client's version number for the file and the server's version number indicates that the conflict has occurred. In this case, the original client's reopen of the remote file object descriptor fails.

Another cause for conflict is that a file may get corrupted by a bad server crash. In this case, the server's disk scavenging program increments the file's version number, and this will prevent any clients from recovering I/O streams to that file.

The disk scavenger may also find files that are no longer referenced by any directory, although this does not cause a conflict or an error. This situation can arise because of the semantics of the UNIX REMOVE operation. It is possible to remove a directory entry for a file that has an open I/O stream to it, and the file is not deleted until the I/O stream is closed. If the server crashes or reboots with files in this state, then the disk scavenger does not delete them (it puts them into a ''lost+found'' directory) and it is still possible for clients to reopen I/O streams to these files. The most common way this case arises is when new versions of programs are installed. The previous version of the program image is removed by name, but it may linger on disk until all instances of the program have terminated.

6.4. Delayed Writes and Failure Recovery

While the recovery system maintains the server's caching state across failures, the use of write-back caching does introduce some windows of vulnerability in Sprite. This section reviews these problems, and describes the steps taken to mitigate them.

First and foremost, using delayed writes raises the possibility that a client failure can cause some recent data to be lost. The length of the delay trades off reliability for better performance. The longer the delay, the greater the chance that a write-back will be avoided because dirty data will be overwritten or deleted. Measurements in Chapter 8 indicate that as much as 50% of the data generated by Sprite clients does not get written back when using a 30-second delay policy. However, a longer delay increases the chance that data will be lost in a system failure. To mitigate this danger, Sprite provides a system call that explicitly forces a range of file blocks to disk and returns only after the data has been safely written out. This call is used by our editors and source code control programs. The Sprite kernel also attempts to write-back the cache upon a system error. This write-back cannot be counted on 100%, however, because a bug in a critical part of the cache module could foil a write-back. In the worst case (e.g., a power failure on the client), a user loses 30 seconds of his work. This is the same guarantee made by a standalone UNIX system, and we view this as an acceptable trade off against the higher performance we obtain during normal use.

A server failure does not necessarily cause loss of data, although the risk is dependent on the writing policy used by the client and the server. To prevent loss of data, the client must retain dirty cache blocks until they are safely on the server's disk. If the server crashes, the data is either on disk or still in the client's cache. With this write-back policy, the server must write each block to disk as it arrives from the client. However, this writing policy can require as many as 3 disk I/Os for each block: one for the data, one for an index block, and one for the file descriptor.

In practice we use a more efficient writing policy that has a small window of vulnerability [Nelson88b]. Servers wait until all the blocks for a file arrive from a client before writing them to disk.[17] This amortizes the writes of the index blocks and the file descriptor over the cost of writing the whole file. Unfortunately, in the current implementation the client is unaware of the server's writing policy. A client assumes that when a WRITE RPC completes it is safe to discard the cache block. Thus, it is possible to lose data that has been written from the client cache to the server cache but has not yet made it to the server's disk.

The best solution to this problem would be to change the cache write-back mechanism from a block-oriented to a file-oriented system. A client could retain a whole file, or a large portion of a big file, until the server has it safely on its disk. This write-back policy would allow amortization of the disk I/O's for index blocks and descriptors, while still allowing a client to age blocks in its cache before writing them out. A file-oriented write-back policy would also work well with the log-structured file system currently being developed for Sprite [Ousterhout89a].

Another improvement to the current implementation would be the addition of atomic file replacement on the server. With atomic replacement, a new version of the file would be committed only after all the blocks for it have arrived from a client and been written safely to disk. A single disk I/O could be used to commit the version, typically by writing a new version of the file descriptor. LOCUS implements atomic replacement in this way. Atomic replacement means that if the server crashes with some of a file's blocks still dirty in its cache, the file will revert to the previous version after the server recovers. Atomic replacement also means that if a client crashes after writing back only part of a file then the server can retreat to the previous version. One problem with atomic

[17] Our servers use the write-through-on-last-dirty-block policy described in [Nelson88b]. The servers can also be configured to use write-through for best reliability or full-delay for best peformance.

write schemes concerns very large files. Extra disk space is needed until the new file version is committed. Another problem is that atomic replacement is not appropriate for database-like files that are updated by changing individual records within the file.

The final problem introduced by delayed writes stems from network partitions. As described above in Section 8.3.2.3, a partition can result in clients generating conflicting versions of a file. A conflict is detected by the server's cache consistency algorithm when the original client tries to recover the server's state. The original client's version number will be out-of-date, so its recovery attempt for the file will be denied by the server. Currently there is no way for the client to save its conflicting version.[18] This conflic is rare, however. Data presented in Chapter 8 shows that most of the write-shared files are not cached so partitions are not a problem, and those files that are cached are not shared so the client can recover successfully.

6.5. Operation Retry

In order to make error recovery transparent to applications, operations that fail due to a server failure should be retried automatically after the state recovery protocol completes. However, there are some cases in which it is better to abort an operation than wait for recovery. Typcially these situations arise because users do not want to wait for recovery; they would rather abort what they are doing and try to get some work done with a different server. There are also status-related operations in which an immediate error reply is more appropriate than waiting for the server to come back on-line. Thus, operation retry should be automatic by default, yet it should also be possible to abort operations that are waiting for recovery.

One approach to operation retry is to build it into the communication protocol. While the Sprite RPC will system resend a packet several times if needed, it eventually gives up and returns a timeout error. The NFS file system, in contrast, uses an RPC protocol with an indefinite retry period[RPC85]. The SunOS RPC protocol used with NFS simply retransmits a request until the server responds, even if the client must wait minutes or hours for the server to come back on-line.[19] This approach relies on the stateless nature of the NFS file servers. The NFS server can handle any RPC request after it boots because it does not keep any open file state. However, retrying an operation doggedly until it succeeds may not be what the user desires. Instead, the user may prefer to abort the operation and continue work with another available server. Unfortunately, the SunOS RPC protocol is uninterruptible, so a user can get stuck attempting an operation with a crashed NFS server.

---

[18] The CODA file system [Satyanarayanan89] adds a special directory structure called a *covolume* as a place to store conflicting versions.

[19] It is possible to use finite timeout periods with the SunOS RPC protocol. However, the failure semantics of NFS are not well defined, so in practice RPC connections to NFS servers do not use a timeout period.

In Sprite, operation retry is handled at a higher level. After an RPC timeout, control is returned to high-level procedures that are above the object-specific naming and I/O interfaces. At this level a process either waits for the completion of the recovery protocol and retries, or it aborts the operation. Operations are structured this way for two reasons. First, the semantics of the operation determine whether or not it should be retried at all. It may be more appropriate to communicate a failure status to the user than it is to force the user to wait for the server to recover. The operations that are not retried are described below. Second, this approach means that no object-specific resources are held locked while waiting for the recovery protocol. Thus, it is possible for a user to abort an operation that is waiting for recovery, just as a blocked I/O operation can be aborted. This high-level approach to operation retry gives the system (and users) more flexibility in the way failures are handled.

There are three responses to the failure of a remote operation: retry them after server recovery, invoke special error handing, or reflect the error up to the application. These cases and the operations that they apply to are discussed below.

6.5.1. Automatic Retry

The main category or operations are those that are automatically retried after recovery. They include all the naming operations (e.g., NAME_OPEN, REMOVE, REMOVE_DIR, MAKE_DIR, MAKE_DEVICE, HARD_LINK, RENAME, SYM_LINK, GET_ATTR and SET_ATTR) and most of the I/O operations (e.g., READ, WRITE, IOCTL). The retry loop for the naming operations is built into the iterative lookup procedure described in Chapter 4, and the retry loop for I/O operations is built into the retry loop associated with blocking I/O operations. In these cases, the additional complexity introduced by failure recovery was minimal.

The user is informed when an RPC timeout occurs, and they have the option of aborting the current operation via a keyboard interrupt (control-C). Kernel error messages are routed to the system error log that is typically displayed on a window of the user's workstation. A user can then abort the operation that caused the failure, or the user can wait for the operation to be retried after recovery.

6.5.2. Special Handling

The second catagory of operations are those that are not retried directly, but cause some special error handling to occur. This action is appropriate for remote operations that are one part of a larger operation. Important examples include the recovery protocol itself, notifications from I/O servers, cache consistency callbacks, and the SELECT RPC that polls the status of a remote object. These cases are discussed below.

The simplest approach to handing an error during the recovery protocol is to abort it and retry the recovery protocol later. This approach assumes that recovery has failed because the server has crashed again. It is also possible that the heavy load from the recovery protocol has overloaded the server so much that it cannot service RPCs. This case has occured because of deficiencies in the RPC implementation, as described below in Section 6.7. In either case, the idemponent nature of the recovery protocol means it is always okay for a client to abort the protocol and restart it later.

The WAKEUP RPC is not retried. This RPC is used by I/O servers to notify a process that is blocked on an I/O operation. If a WAKEUP fails it means the client has crashed, in which case the notification is unnecesary, or that there has been a network partition. In this case, the server can rely on the fact that the client will invoke the recovery protocol when the partition ends and then retry any I/O operations. The I/O operations are retried after recovery no matter what caused them to block in the first place so there is no danger from lost wakeups.

The cache consistency callbacks from the file server to a client are not retried so that a failed client does not hold up operations by different clients. An OPEN, for example, may trigger callbacks to other clients, and the failure of one client to respond should not prevent the OPEN from completing. Instead, the server invokes the CLIENT_KILL procedure to forcibly close I/O streams from the unresponsive client.

The SELECT operation is not retried because a process is generally polling several file system objects. In this case, waiting for recovery on one object would prevent the polling of other objects. Unfortunately, it is difficult to communicate a useful error message back to an application via the UNIX **select** interface. It would be best to return an error status from **select** and leave the bit set in the select mask that corresponds to the I/O stream that incurred the timeout. However, UNIX applications do not understand this convention; they usually assume an error from **select** is a disaster and they abort. In the current implementation the error is masked and **select** indicates that the offending stream is ready for I/O. This causes the application to **read** or **write** the stream and then block awaiting recovery. This approach is not perfect, however, and we may change **select** to mask the error altogether so the applications just ignore the stream.

6.5.3. No Retry

The third catagory of operations are those that are not retried and cause an error to be returned to the user if the server is unavailable. This approach is appropriate for status-related commands such as the DOMAIN_INFO RPC that returns the amount of free space on a disk. Similarly, the attributes operations on open I/O streams are not retried. An application to query the attributes of its I/O streams to detect failures. There are other cases in which failed operations are not retried, although the absense of retry can arguably be considered bugs in the implementation. These are discussed below.

The IO_OPEN RPC is not retried. A remote device open will fail instead of being retried automatically. This particular case would be relatively straight-forward to fix if it became a problem.

The CLOSE RPC is not retried. A process can close an I/O stream after the server fails and not get hung up during the close. The good example is that changing a process's working directory is implemented by opening an I/O stream to the new directory and then closing the I/O stream to the previous directory. If the CLOSE operation blocked waiting for recovery, a process would not be able to leave the domain of a failed server. Note that this approach can introduce inconsistencies during a network partition (the server will still think the stream is open), although the recovery protocol will repair the damage.

The RPCs used during the migration of I/O streams are not retried. Failure of a host during migration can destroy an I/O stream. However, the process manager also aborts a process if a host fails during migration. Ideally the migration implementation could back out if the destination client fails, but this has not been implemented.

These examples highlight the fact that error recovery in not always clean and simple. It is tempting to simplify error handling by taking one of two approaches. One approach is to abort the current operation and reflect the error up to the application. This approach is not always appreciated by users. The second approach is to retry operations indefinitely in the low-level communication protocol. However, this approach is not flexible enough to handle more complex situations. In Sprite, the low-level communication mechanism will time out, but higher-level system software does its best to recover from the failure.

6.6. Crash and Reboot Detection

The recovery system depends on a low-level *host monitor* module that monitors network communication in order to detect host failures. There are two main issues involved with failure detection. The first is that failure detection is an uncertain task; it is not possible to distinquish between a host failure and a network failure. Both of these cases are manifested as an inability to communicate with the other host. The second issue is the cost of the failure detection system. Failure detection adds overhead because network traffic must be monitored, and additional network traffic is generated in order to verify that other hosts are still alive. The failure detection system is described below, and then measures of its cost are presented.

The host monitor reports two events: crashes and reboots. Crashes and reboots are distinguished because servers need to clean up after their clients crash, while clients need to initiate recovery action after their servers reboot. The interface to the host monitor module is via *callback procedures*. Other kernel modules register procedures to be called in the event of a crash or a reboot (different procedures are registered for crashes and reboots).

The host monitor detects crashes by monitoring the RPC system for communication failures. The RPC system will resend a request message several times before giving up and reporting a timeout error. An RPC timeout indicates that the host is down or the network is partitioned, and this failure triggers the crash callbacks so that other kernel modules can clean up. The fact that network partitions and host failures are treated the same means that a server could close I/O streams associated with a partitioned client. After the partition, however, the client will invoke the recovery protocol in an attempt to reopen these streams. However, as described in Section 6.3.2.3, there is a possibility of conflict in recovery after a partition.

One way to detect reboots is to periodically poll (or ‘‘ping’’) another host at a high enough frequency that it cannot crash and reboot without causing a communication failure. However, the cost of this solution could be high. It requires clients to generate network traffic to the server even when they are otherwise idle. Sprite uses a different technique that relies on a generation number in the RPC protocol packet header. The generation number is set to the time a host boots up. The time is obtained from a battery powered clock, or from a special RPC that broadcasts a request for the time. The host monitor examines every arriving packet to see if the peer host's generation number has changed, indicating a reboot.

With this technique, it is only necessary to ping another host in order to bound the time it takes to detect a crash or reboot. The rate of pinging, and the hosts which are pinged, can be tuned to reduce the system load. For example, file servers do not care if a client crashes unless that client holds some resource. The normal callbacks associated with caching and file locking are sufficient to detect when a client has crashed while holding a resource. A timeout on a callback triggers the crash callbacks that clean up after the client. Thus, there is no need for servers to periodically check up on their clients. Clients, however, ping their servers in order to bound the time it takes to detect a reboot. Their periodic check ensures that clients initiate the recovery protocol in a timely fashion.

The host monitor pings hosts for which reboot callbacks are registered, so other kernel modules need not do their own pinging. Reboot callbacks can be unregistered so that the host monitor does not have to continue pinging another host forever. As an example, the process manager registers a reboot callback on a remote host that is hosting a migrated process, and it cancels the callback after the process terminates. The host monitor also optimizes its pinging by suppressing a communication attempt if there has been recent message traffic from the other host.

The network-wide cost of pinging is proportional to R * C * S, where R is the rate of pinging, C is the number of clients, and S is the number of servers. This cost assumes the worst case where all clients are interested in all servers, and it ignores the effect of ping suppression due to recent message traffic. In contrast, a naive implementation in which all hosts actively monitor all other hosts would have a cost proportional to R * N * N, where N is the total number of hosts. C * S will be much smaller that N * N if S is smaller than C, which is true in a large network of diskless workstations.

Periodic checking, noting RPC timeouts, and noting changes in the RPC boot generation number are sufficient to differentiate two states for a host, ‘‘alive’’ and ‘‘dead.’’ A ‘‘booting’’ state is also needed in order to smooth recovery actions. Servers may go through a significant amount of setup and consistency checking before they are available for service. For example, our servers take about 5 minutes per disk to verify its directories and file maps. During this time a server may be talking to the outside world, but it may not be ready for client recovery actions yet. Without the intermediate booting state, servers appear to flip between being alive and dead as they come up and generate network traffic, but refuse to service requests. The RPC protocol supports the booting state by marking out-going requests with a ‘‘non-active’’ flag, and returning a ‘‘non-active’’ error code to any request. The host monitor module sees this error return and does not report a reboot until a host is fully alive.

6.6.1. Measured Costs of Crash Detection

Monitoring RPC traffic adds some overhead to the RPC protocol. All incoming messages must be checked for a change in boot generation number. All timeouts also have to be noted. On Sun-3/75 workstations, this increases the average cost of a Sprite kernel-to-kernel echo RPC from 2.19 msec to 2.45 msec. This is a rather large overhead on a null call, about 10%, although it only adds about 3.6% to the cost of sending a 4 Kbyte block via RPC (7.1 msec vs 7.36 msec). This overhead is time required to acquire a monitor lock, hash to find the host's state, read the clock to record the time at which the current message arrived, and verify that the host has not changed state. The overhead is incurred twice per RPC, once when the server receives the request message, and once when the client receives the reply message. This cost could probably be reduced by careful coding.

The background cost due to pinging other hosts is quite small. The cost was determined by measuring a network of Sun3s with 2 servers and 15 clients during a period of no other system activity. CPU utilization was measured by counting trips through the kernel's idle loop, and comparing this against an idle calibration done during the boot sequence. The clients ping their servers every 30 seconds with a null ECHO RPC. Each client host was found to consume a very small fraction of the server's CPU, about 0.0036%. No applications were executing on the clients, so the load was only due to the ping traffic from the clients. This load correlates well with the estimated load of a single RPC done every 30 seconds. If half the cost of an RPC, 2.45 msec, is charged to the server's CPU, then one RPC every 30 seconds consumes .001225/30 CPU seconds, which is about 0.004% utilization. In reality, less than half of the elapsed time for an ECHO RPC is consumed by the server's CPU. There is some time spent on the network itself and in the network interface hardware. Working back from the measured CPU load, 0.0036% per ECHO, we can estimate that about 1 msec of the 2.4 msec is spent in the server's CPU. This time includes interrupt handling and a process switch to a kernel RPC server process.

Another way to measure the effect of pinging on the servers is to compare the number of echo RPCs serviced in comparison with other RPCs. Figure 6-2 shows the number of RPCs serviced by three of the file servers over a 6 month period. The echo traffic is basically constant throughout the day because it is a function of the number of clients a server has. Overall, about 6% of the servers' RPC traffic is due to pings, but this traffic only accounts for 2% to 4% of the day-time traffic. Note that Mint has substantially less echo traffic than Allspice and Oregano, but each of these servers is shared by all clients. The lesser echo traffic on Mint reflects an optimization that suppresses a ping if there has been other recent message traffic between the hosts. Clients ping every 30 seconds, and a ping is suppressed if there has been another message within 10 seconds. Suppression is effective on Mint because once a minute each client updates a host status database that is stored on Mint. However, the 10 second value for ''recent'' is overly conservative for it to be effective on the other servers. A value for ''recent'' that is greater than or equal to the ping interval would suppress many more pings while the server is up, yet it would still cause the clients to ping at the same rate (i.e., every 30 seconds) once the server crashed. Thus, clients would still detect the server reboot

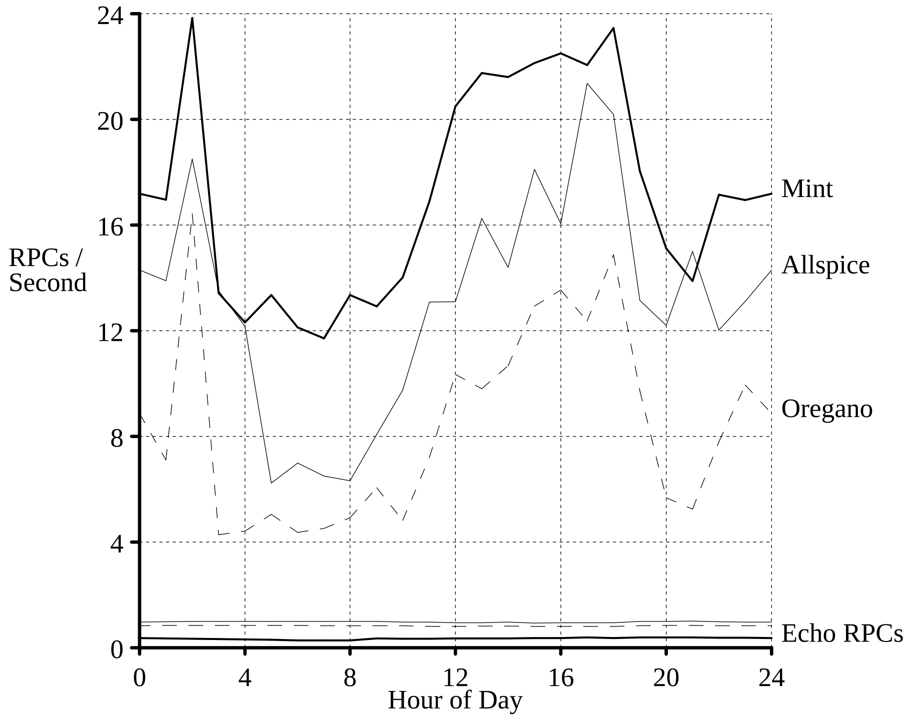


Figure 6-2. Echo traffic on three file servers from July through December 22, 1989. The upper line for each server is its total RPC traffic. The three lines labeled ''Echo'' are the echo RPC traffic for each server.

relatively quickly, and the echo traffic could be reduced even further.

6.7. Experiences

The recovery mechanisms described here are real, and they are put to the test rather frequently in our development environment. Table 6-1 lists the number of reboots on a per-month basis for the file servers and three of the clients. The total number of reboots are shown, and then the number of different days on which reboots occurred is given in parentheses. These reboots occurred for routine kernel upgrades or because the system crashed. (Unfortunately these two cases are not distinguished in the reboot logs.) The clients are included to highlight the difference between a Sprite developer (Sage) and a normal Sprite user (Basil). Rebooting is common with developers as new kernels are tested. (Ideally a developer has two machines so he does not have to reboot his workstation, but this is not always the case.) Basil's user, on the other hand, only reboots his workstation after a crash or if we insist that its kernel be upgraded to fix some bug. Mace was used for kernel testing in June and July, and for regular work the rest of the time.

Reboots per Month							
Month	Mint	Oregano	Allspice	Assault	Basil	Sage	Mace
Jan	25 (12)	21 (15)	- -	- -	12 (6)	56 (17)	17 (6)
Feb	20 (12)	20 (12)	- -	- -	12 (11)	77 (17)	16 (14)
Mar	35 (17)	38 (18)	- -	- -	8 (7)	43 (18)	17 (11)
Apr	25 (12)	25 (14)	- -	- -	3 (3)	53 (10)	7 (5)
May	45 (19)	25 (14)	- -	- -	10 (9)	22 (13)	20 (12)
Jun	24 (12)	20 (13)	- -	- -	12 (7)	49 (10)	53 (15)
Jul	9 (7)	14 (10)	7 (3)	0 (0)	1 (1)	80 (13)	61 (13)
Aug	29 (12)	18 (14)	24 (12)	24 (5)	5 (5)	29 (18)	7 (5)
Sep	18 (12)	8 (7)	15 (11)	30 (9)	3 (3)	27 (7)	7 (5)
Oct	21 (14)	8 (6)	22 (13)	13 (10)	2 (2)	15 (8)	5 (4)
Nov	31 (16)	11 (9)	15 (10)	12 (8)	4 (4)	11 (6)	7 (3)
Dec	7 (4)	14 (9)	4 (3)	2 (2)	1 (1)	3 (3)	3 (2)

Table 6-1. Reboots per month for the file servers and three clients. The first number for each host is the total number of reboots, and the number in parentheses is the number of different days on which a reboot occurred. Sage is used by a Sprite developer and reboots frequently because of kernel testing. Basil's user, on the other hand, reboots as infrequently as possible.

The system's stability has been improving throughout the year, and there have been long periods without server failures. Table 6-2 gives the mean time between reboots (MTBF), the standard deviation, and the maximum uptime observed, for the file servers and three clients. This was computed from July 1989 through January 1990. While the MTBF of our servers is still only a few days, it is improving steadily as we continue to find and fix bugs. The long maximum uptimes suggest that the servers are basically

Mean Time Between Reboots (Days)			
Host	Average	Std Dev	Max
Mint	1.63	2.63	21.69
Oregano	2.56	3.29	20.96
Allspice	2.09	3.12	15.93
Assault	1.92	4.12	22.90
Basil	11.59	9.95	43.42
Sage	1.27	4.03	41.40
Mace	2.16	3.91	16.20

Table 6-2. The average time between reboots, its standard deviation, and the maximum observed uptimes. Values are in days. This was computed for the interval from July, 1989 through January, 1990.

stable, until we introduce new bugs as part of our research!

Ironically, the state recovery protocol places such a high load on the servers that it has exposed several bugs that lurked undetected for years, and only showed up after the system grew large enough. Races in data structure management routines were exposed, and bugs in the RPC protocol were also flushed out. Perhaps the most glaring problem is a case of starvation in the RPC channel allocation code. It is possible for a subset of the clients to occupy all of a server's RPC channels while they perform a series of RPCs to reopen their descriptors. During this time other clients experience timeout errors that cause them to invoke their recovery protocol again. This effect has been dubbed a ''recovery storm''. Clients often go through recovery 2 or 3 times after a server boots, and in one case recovery failed altogether because no client could complete its recovery protocol. This failure was due to a bug in the host monitor module. Ordinarily it ensures that only one process is doing the file system recovery at a time. A bug allowed multiple processes on the same client to do recovery simultaneously, and this increased the load on the server enough to cause a total failure.

We are currently investigating further refinements to the recovery protocol to reduce the load on the server. Mary Baker is studing the RPC system during recovery storms and will be tuning the RPC system in order to avoid this problem. The main defect in the RPC protocol is the lack of a negative acknowledgment for the case when the server has no RPC channels left for a new request. (Appendix C has a complete description of the RPC protocol.) Clients give up and report a timeout instead of backing off and retrying later. We can also refine what descriptors are reopened during recovery as noted in Section 6.3.2.2. Finally, instead of a single RPC per REOPEN, the clients could batch their requests.

6.8. Conclusion

This chapter has presented a recovery system for stateful servers based on the simple principal of keeping redundant state on the clients and servers. The servers keep state in main memory instead of logging it to disk, and they rely on their clients to help them rebuild their state. This is done via an idempotent state recovery protocol that is initiated by the clients whenever they detect the server's state might be inconsistent with their own. The server attempts to reconcile its state with that of the client, but the server may detect conflicts and force the client to change its state. In the case of the Sprite file system, a conflict might prevent an I/O stream from being successfully reopened, but these cases are rare in practice.

Overall the experience with the recovery system has been positive. Sprite has been running nearly continuously in our network for over two years, although there have been many individual host failures and a number of network partitions. Depending on what a user is doing he may or may not notice a server reboot. Only if the server is needed for some resource will a user be delayed until after the recovery protocol. In many cases, too, the user has the option of aborting an operation that is waiting for recovery, so that he can continue with other work.

CHAPTER 7

Integrating User-level Services

7.1. Introduction

A common approach in modern operating systems is to move system services outside the operating system kernel. Debugging is easier because the service is implemented as an ordinary application and the standard debugging tools apply to it. The kernel remains smaller and more reliable. In addition, it is easier to experiment with new types of services at user-level than by modifying the kernel. These advantages of user-level implementation of system services have been promoted before by designers of message-based kernels [Cheriton84] [Accetta86]. However, the main drawback of user-level services is the performance penalty that comes from message passing and context switching overhead. There is a conflict between having a high-performance, kernel-resident service, and an easy-to-manage user-level service. In Sprite, this conflict is addressed by a hybrid approach that puts performance-critical services (e.g., the file system) inside the kernel and that provides a way to cleanly extend the system with additional user-level services.

In Sprite, user-level services appear as part of the file system; the permits them to take advantage of the distributed name space and remote access provided by the file system architecture. A *pseudo-device* is a user-level server process that implements the file system's I/O interface. A pseudo-device server registers itself as a special file, and the kernel forwards all I/O operations on the file to the user-level server process. The mechanisms that support remote device access also support remote access to pseudo-devices. A *pseudo-file-system* is a user-level server process that implements the naming interface (and usually the I/O interface, too). A pseudo-file-system server registers itself as a prefix in the name space. The prefix table mechanism integrates the pseudo-file-system into the distributed name space, and the kernel forwards all naming operations on that domain to the user-level server process.

The advantage of using the file system interface to access user-level services is that it is a natural extension of the device-independent I/O already present in UNIX. The **write** and **read** operations are equivalent to the send and receive operations of a message-based system. The **ioctl** operation is equivalent to a synchronous send-receive pair of operations, and it provides a simple RPC transport protocol between client applications and the user-level server process. The server is free to define any **ioctl** commands it wants to support, so a variety of services can be implemented. Furthermore, **select** also applies to the user-level services, so it is possible to wait for devices and services simultaneously. In addition to using these familiar I/O operations, the Sprite file system also provides a distributed name space and transparent remote access. By integrating services into the file system they are automatically accessible throughout the network. There is no need to invent a new name space for new services.

Communication between the kernel and the user-level service is via a *request-response* protocol that is much like RPC. The kernel passes a request to the user-level server process and waits for a response. The protocol uses a buffering system that allows batched reads and writes in order to reduce the number of context switches required for kernel-to-user communication. Message buffers for the protocol are in the server process's address space to give the server control over the amount of buffer space and to make it possible to release idle buffer pages. Benchmarks of the request-response protocol indicate that communication with a user-level server is about as fast as using the UNIX pipe mechanism locally. It is much faster in the remote case than the UNIX TCP protocol because it uses the Sprite network RPC protocol. However, the overhead of the protocol is high enough to slow I/O-intensive tasks by 25% to 50% in comparison to a kernel-resident service.

In Sprite, there are currently four main applications for user-level servers: an X11 window system server, a terminal emulation package, a TCP/IP protocol server, and an NFS file system server. The implementation is fast enough to provide good interactive response in the window system, even for things like mouse tracking. However, the performance of our TCP/IP protocols is not as good as the kernel-resident UNIX implementation of these protocols, as expected. The performance penalty is similar to that observed by Clark with his upcall mechanism [Clark85], which is used to implement network protocol layers outside the kernel in user code. The performance of NFS access from Sprite is also affected by the user-level implementation. The penalty for compilation benchmarks is about 30% in comparison to a native NFS implementation. The conclusion regarding performance is that it is acceptable for many applications, but the penalty is significant enough that heavily-used services, especially regular file access, are worth implementing in the kernel.

This chapter is organized as follows. Section 7.2 describes the major applications for pseudo-devices and pseudo-file-systems that exist in Sprite. Section 7.3 reviews the file system architecture and describes how pseudo-file-systems and pseudo-devices are implemented. Section 7.4 describes the request-response protocol between the kernel and the server process. Section 7.5 evaluates the performance of the implementation, from the raw performance of the request-response protocol up to a high-level benchmark executed on the NFS pseudo-file-system. Section 7.6 describes related work done in other systems. Section 7.7 concludes this chapter.

7.2. User-level Services in Sprite

The X window server is implemented as a pseudo-device. The X server controls the display and multiplexes the mouse and keyboard among clients, as shown in Figure 7-1. The clients use **write** to issue commands to the X server, and **read** to get mouse and keyboard input. A buffering system, which is described in detail in Section 7.4.2, provides an asynchronous interface between the window server and its clients to reduce context

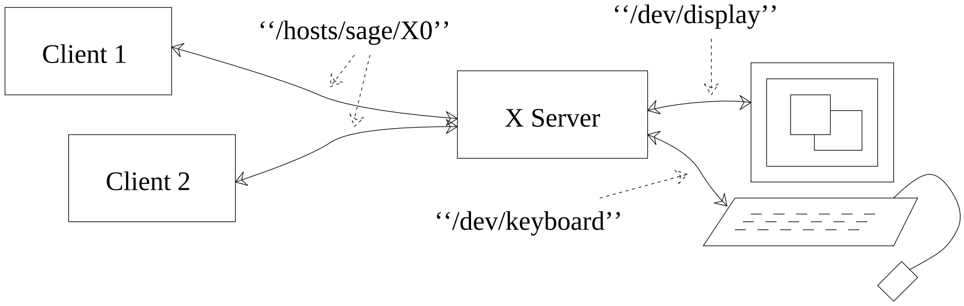


Figure 7-1. The pseudo-device ''/hosts/sage/X0'' is used by clients of the window system to access the X window server on workstation ''sage.'' The server, in turn, has access to the display and keyboard.

switching overhead. There is a pseudo-device for each display in the network, and access to remote displays is not a special case because the file system provides network transparency.

The TCP/IP protocols are used by Sprite applications to interface to non-Sprite systems for mail transfer, remote logins, and remote file transfer. These applications do not necessarily demand high-performance, and so we implement the TCP/IP protocols at the user-level as a pseudo-device. Client processes read and write a pseudo-device to use TCP, and the user-level server process implements the full TCP protocol by reading and writing packets over the raw network. The internet server defines **ioctl** operations that correspond to the 4.3BSD socket calls such as **bind**, **listen**, **connect** and **accept**. The **ioctl** input buffer is used to pass arguments from the client to the server, where the socket procedure is executed. The **ioctl** output buffer is then used to return results back to the client. Each socket library procedure in the client is simply a stub that copies arguments and results into and out of buffers and invokes the **ioctl**. In this case, the pseudo-device mechanism provides the transport mechanism for an RPC-like facility between clients and the internet server.

Terminal emulators are implemented as pseudo-devices. A terminal emulator provides terminal-like functions such as backspace and word erase for non-terminal devices like windows or TCP network connections. Client processes make **read** and **write** requests on the pseudo-device as if it were a terminal. The server implements the client's requests by manipulating a window on the screen or a TCP connection to a remote host. The server provides the full suite of 4.3 BSD **ioctl** calls and line-editing functions such as backspace and word erase. In this case, the pseudo-device mechanism provides a generalization of the 4.3BSD pseudo-tty facility.

A pseudo-file-system server is used to provide access to NFS file systems from Sprite. The pseudo-file-system server translates Sprite file system operations into the NFS protocol and uses the UDP datagram protocol to forward the operations to NFS file servers. The NFS pseudo-file-system server is very simple. There is no caching, of

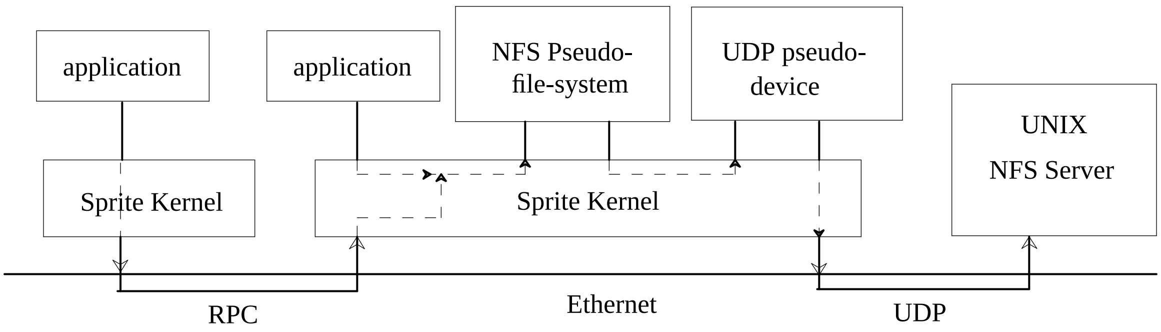


Figure 7-2. Two user-level servers are used to access a remote NFS file server. The first is the NFS pseudo-file-system server. In turn, it uses the UDP protocol server to exchange UDP packets with the NFS file server. The figure also depicts requests to the NFS pseudo-file-system server arriving over the network from remote Sprite clients using the Sprite network RPC protocol. The arrows indicate the direction of information flow during a request.

either file data or file attributes, and the server process is single-threaded. In this case, our goal was to provide NFS access to our users with minimum implementation effort, not necessarily maximum performance. Most files are serviced by higher-performance Sprite servers, and NFS is used to access the files left on NFS servers.

Figure 7-2 illustrates the communication structure for NFS access under Sprite. Note that the UDP network protocol, which is used for communication between the pseudo-file-system server and the NFS server, is not implemented in the Sprite kernel. Instead it is implemented by the internet protocol pseudo-device server. This approach adds additional overhead to NFS accesses, but it illustrates how user-level services may be layered transparently. The interface between NFS and UDP would not have to change if the UDP protocol implementation were moved into the kernel. Figure 7-2 also shows an application accessing the NFS pseudo-file-system from a Sprite host other than the one executing the pseudo-file-system server. In this case the kernel's network RPC protocol is used to forward the operation to the pseudo-file-system server's host.

7.3. Architectural Support for User-level Services

The operating system architecture is organized in a modular way to support access to different kinds of objects, including pseudo-file-systems and pseudo-devices. The architecture includes two major internal interfaces, one for the naming operations listed in Table 7-1, and one for the I/O operations listed in Table 7-2. Allowing a user-level server process to implement these interfaces requires the addition of a communication mechanism between the kernel and the server process. A request-response style protocol is used so that the interactions between the kernel and the server are similar to those

Pseudo-File-System Operations	
Open	Open an object for further I/O operations.
GetAttr	Get the attributes of an object.
SetAttr	Set the attributes of an object.
MakeDevice	Create a special device object.
MakeDirectory	Create a directory.
Remove	Remove an object.
RemoveDirectory	Remove a directory.
Rename	Change the name of an object.
HardLink	Create another name for an existing object.
SymbolicLink	Create a symbolic link or a remote link.
DomainInfo	Return information about the domain.

Table 7-1. Naming operations that are implemented by pseudo-file-system servers, and the DomainInfo operation that returns information about the whole pseudo-file-system.

Pseudo-Device Operations	
Read	Transfer data from an object.
Write	Transfer data to an object.
WriteAsync	Write without waiting for completion.
Ioctl	Invoke a server-defined function.
GetAttr	Get attributes of an object.
SetAttr	Set attributes of an object.
Close	Close an I/O connection to an object.

Table 7-2. I/O operations on a pseudo-device or an object in a pseudo-file-system. I/O operations on the object are forwarded to the server using the request-response protocol.

between two Sprite kernels. Stub procedures that use this protocol are invoked through the internal interfaces, just as stub procedures that use the network RPC protocol are invoked in the remote case. Thus, the addition of user-level servers adds a third, orthogonal case to the basic system structure, in addition to the local and remote cases discussed in Chapter 3.

A user-level server process implements the I/O interface for a pseudo-device, while a Sprite file server implements the naming interface for the pseudo-device. A special file is used to represent the pseudo-device in the name space, just as a special file is used to represent a device. Thus, all naming operations on the pseudo-device are handled by a Sprite file server. For the I/O interface to pseudo-devices, the kernel uses a request-response protocol to forward the I/O operations to the user-level server, and the server is free to implement the I/O operations however it wants. Remote access to a pseudo-device is handled just like remote device access. The network RPC protocol is used to forward the operation to the host running the pseudo-device server (the *I/O server* for the pseudo-device). At the I/O server the operation is forwarded to the user-level server process with the request-response protocol.

With a pseudo-file-system, a user-level server process implements the naming interface, and the prefix table mechanism automatically integrates the pseudo-file-system into the distributed file system name space. The pseudo-file-system server registers itself as a domain in the prefix table of its host, just as the kernel registers domains for any local disks. The prefix for the pseudo-file-system is exported to the network in the same way that prefixes corresponding to local disks are exported. To remote clients, the pseudo-file-system is accessed with the same remote naming module it uses to access remote domains implemented by Sprite file servers. Thus, there are three cases implemented below the internal naming interface: 1) the server for a prefix can be the local kernel, 2) a remote kernel, or 3) a user-level process.

A pseudo-file-system server has the option of implementing the I/O interface as well as the naming interface. The NFS pseudo-file-system server, for example, implements both interfaces because all file system operations have to be forwarded to the NFS file server. However, because the I/O interface is independent of the naming interface, the pseudo-file-system server can arrange for the kernel to implement the I/O interface to objects in a pseudo-file-system. In this case the object would be a file, device, or pipe, but the pseudo-file-system would provide its own naming interface to the object. The implementation of described below.

7.4. Implementation

The server for a pseudo-device or pseudo-file-system is much like the server in any RPC system: it waits for a request, does a computation, and returns an answer. In this case it is the Sprite kernel that is making requests on behalf of a client process, and the server is a user-level application process. The kernel takes care of bundling up the client's parameters, communicating with the server, and unpackaging the server's answer so that the mechanism is transparent to the client. The following sub-sections describe the implementation in more detail, including the I/O streams used by the server, the request-response protocol between the kernel and the server, and a buffering system used to improve performance.

7.4.1. The Server's Interface

A pseudo-device server process has one *control stream* used to wait for new clients, and one *request stream* for each **open** by a client process. The control stream is created when the pseudo-device server opens a pseudo-device file. The server distinguishes its **open** from future opens by clients by specifying the O_PDEV_MASTER flag. The request streams are created by the kernel each time a client opens the pseudo-device. The kernel enqueues a message on the control stream that identifies the new request stream, and the server reads the control stream for these messages. All further operations between the client and the server take place using the request stream.

The internal representations of these streams use three types of object descriptors that correspond to the control stream (PDEV_CONTROL), the server's end of a request

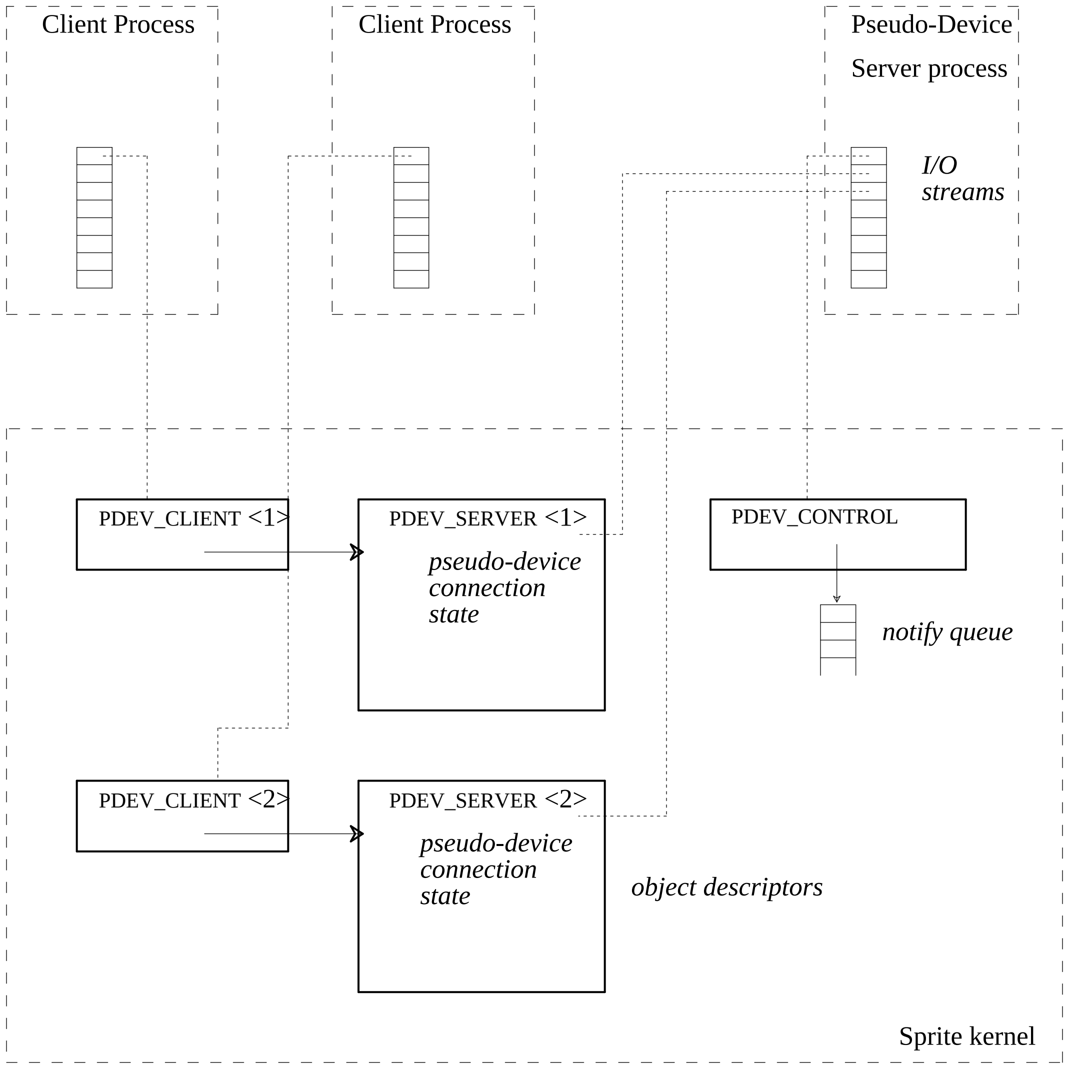


Figure 7-3. I/O streams between clients and a pseudo-device server. There is one pair of kernel object descriptors for each request-response channel between a client and the server. This channel is created each time a client makes an **open** on the pseudo-device. The PDEV_CLIENT descriptor is a stub that references the corresponding PDEV_SERVER descriptor; it is needed so that client I/O operations invoke different object-specific kernel procedures than the server. There is one PDEV_CONTROL stream for each pseudo-device, and this is used to notify the server about new PDEV_SERVER streams.

stream (PDEV_SERVER), and the client's end of a request stream (PDEV_CLIENT). These descriptors are shown in Figure 7-3. There is one PDEV_CONTROL descriptor per pseudo-device. It indicates if a server process is active and it has a queue for the notification messages about new request streams. The PDEV_SERVER descriptor records the state of a connection between the client and the server, and it is used during the request-response protocol described below. The PDEV_CLIENT descriptor is linked to the corresponding PDEV_SERVER descriptor. The client's descriptor is distinct because the type of the object descriptor is used to dispatch through the internal I/O interface, and the client and server have different sets of object-specific procedures. For example, the server reads its end of the request stream to learn about new requests, while a client reads its end of the request data to get input data.

The I/O streams used by a pseudo-file-system server are similar to those of a pseudo-device server, except that the control stream is replaced by a *naming stream*. The naming stream is a request-response stream that the kernel uses to forward naming operations to the server. The naming stream is created when the pseudo-file-system server opens the remote link that corresponds to the domain prefix. A special flag, O_PFS_MASTER flag, distinguishes this open as being by the server process. The client end of the naming stream is attached to the prefix table entry for the pseudo-file-system. This causes pseudo-file-system routines to be invoked through the internal naming interface, and these routines use the naming stream to forward the naming operations to the server.

The pseudo-file-system server has two options when a client makes an OPEN request. If it wishes to implement the I/O operations on the named object, it can ask the kernel to create a new request-response stream. This stream is identical to a pseudo-device request-response stream. Alternatively, the pseudo-file-system server can open an I/O stream to a file, device, or pipe, and ask the kernel to pass this stream off to the client. The system has to be able to pass a stream to a remote client. However, this case is essentially the same problem that process migration causes; an I/O stream must be moved between hosts. The migration mechanism described in Chapter 5 is reused in order to implement stream passing. (While passing off open I/O streams to clients is supported by the kernel, we have not yet developed any major pseudo-file-system applications that use it.)

7.4.2. Request-Response

The Sprite kernel communicates with the server using a request-response protocol. The synchronous version of the protocol is described first, and then extensions to allow asynchronous communication are described. For each kernel call made by a client, the kernel issues a request message to the server and blocks the client process waiting for a reply message. The request message includes the operation and its associated parameters, which may include a block of data. The server replies to requests by making an **ioctl** call on the request stream. The **ioctl** specifies the return code for the client's system call, a signal to return (useful for terminal emulators), and the size and location of any return data for the call (e.g., data being read). The kernel copies the reply data directly from the server's address space to the client's.

7.4.2.1. Buffering in the Server's Address Space

The kernel passes request messages to the server using a *request buffer*, which is in the server process's address space. An associated pair of pointers, *firstByte* and *lastByte*, are kept in the kernel and indicate valid regions of the buffer. There is one request buffer for each request stream (and naming stream) that the server has. With this buffering scheme the server does not read the request messages directly from its request stream. Instead, the kernel puts request messages into the request buffer, and updates lastByte to reflect the addition of the messages. The server then reads a short message from the request stream that has the current values of firstByte and lastByte. The **read** returns only when there are new messages in the request buffer, and a server typically uses **select** to wait on all of its request streams and control (or naming) stream simultaneously. After processing the message found between firstByte and lastByte the server updates firstByte with an **ioctl**.

The motivation for this buffering system is to move the buffer space out of the kernel and to eliminate an extra copy. The kernel copies a request message directly from the client process's address space into the server's request buffer, and it copies a server's reply message directly back into the client's address space. In contrast, kernel-resident buffers like those used for UNIX pipes require data to be copied twice each direction, once from the first process (i.e. the client) into the kernel buffer and once from the buffer to the second process (i.e. the server). The server process is free to allocate a buffer of any size, and the buffers do not have to be tied down in kernel space. We shifted to this buffering scheme after our initial experiences with a prototype implementation that used fixed-size, kernel-resident buffers. We found that there were often many buffers associated with idle request streams that were needlessly occupying kernel space. The Sprite kernel is not demand paged, so these idle buffers occupied real memory. User-space buffering eliminates this problem, and it gives the server the flexibility of choosing its own buffer size.

As a convenience to servers, the kernel does not wrap request messages around the end of the request buffer. If there is insufficient space at the end of the buffer for a new request, then the kernel blocks the requesting process until the server has processed all the requests in the buffer. Once the buffer is empty the kernel places the new request at the beginning of the buffer, so that it will not be split into two pieces. This sequence is shown in Figure 7-3. No single request may be larger than the server's buffer: oversize **write** requests are split into multiple requests, and oversize **ioctl** requests are rejected. If a write is split into several requests, the request stream is locked to preserve the atomicity of the original write. Reads are not affected by the size of the request buffer because read data is transferred directly from the server's address space to the client's buffer.

7.4.2.2. Buffering for an Asynchronous Interface

The buffering mechanism supports asynchronous writes (‘‘write-behind’’) at the server's option. If the server specifies that write-behind is to be permitted for the stream, then the kernel will allow the client to proceed as soon as it has placed a WRITE request in the server's buffer. In enabling write-behind, the server guarantees that it is prepared to accept all data written to the stream; the kernel always returns a successful result to clients. The advantage of write-behind is that it allows the client to make several WRITE requests without the need for a context switch into and out of the server for each one. On multiprocessors, write-behind permits concurrent execution between the client and server. All other requests are synchronous, so the first non-write request forces the client to block and give the server an opportunity to catch up. The X window server, for example, can use write-behind because the X interface is already stream-oriented in order to support batching of requests. Window system clients use **write** to issue requests to the server, and the request-response protocol allows these to be buffered up, or batched, before the server has to respond.

The terminal emulation package also uses write-behind to reduce context switching. This is useful when the terminal stream is line buffered, which is the default for most interactive programs. Without write-behind, line buffering would require a context switch to the terminal server on each line of output. With write-behind many lines of output can be buffered before a context switch is required. Write-behind is especially important if the terminal emulator is a client of the window system. Without write buffering, each line of terminal output would require four context switches: from the client to the terminal server, from the terminal server to the display server, and then two more to return to the client. Write-behind, on the other hand, allows many lines of output to be buffered before requiring these context switches so the display of large amounts of data is efficient.

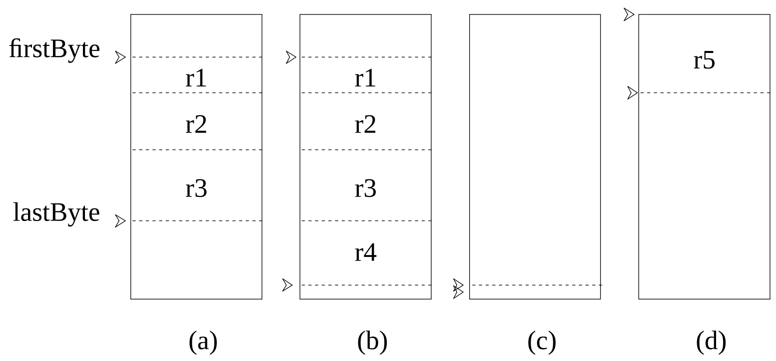


Figure 7-4. This example shows the way the firstByte and lastByte pointers into the request buffer are used. Initially there are 3 outstanding requests in the buffer. The subsequent pictures show the addition of a new request, an empty buffer (the server has processed the requests), and finally the addition of a new request back at the beginning of the buffer.

Read performance can be optimized by using a *read buffer*, one per request stream. The server fills the read buffer, which is in the server's address space, and the kernel copies data out of it without having to switch out to the server process. If the server process declares a read buffer (it is optional), then it does not see explicit READ requests on its request stream. Instead, synchronization is done with firstByte and lastByte pointers as with the request buffer. The server process updates the read buffer's lastByte after it adds data, and the kernel moves firstByte to reflect client reads. The read buffer is used by the X display server, for example. As mouse and keyboard events are generated the server buffers them in the read buffers that correspond to windows. Later on, when the client processes get around to reading their input, the kernel copies the data directly out of the read buffer without paying for a context switch to the server.

7.4.2.3. Buffer System Summary

To summarize the buffering scheme, the server has a request buffer associated with each request stream, and optionally a read-ahead buffer for each stream. These buffers are allocated by the server in its own address space, and an **ioctl** call is used to tell the kernel the size and location of each buffer. The kernel puts request messages directly into the request buffer on behalf of the client. The server's **read** call on the request stream returns the current values of firstByte and lastByte for both buffers. The server updates the pointers (i.e., the request buffer firstByte and read buffer lastByte) by making an **ioctl** on the request stream. The **ioctl** calls available to the server are summarized in Table 7-3.

7.4.3. Waiting for I/O

Normal I/O streams include a mechanism for blocking processes if the stream is not ready for input (because no data is present) or output (because the output buffer is full). To be fully general, pseudo-devices must also include a blocking mechanism, and the server must be able to specify whether or not the pseudo-device is ‘‘ready’’ for client I/O operations. One possibility would be for the kernel to make a request of the server

Server I/O Control Operations	
IOC_PDEV_SET_BUFS	Declare request and read-ahead buffers
IOC_PDEV_WRITE_BEHIND	Enable write-behind on the pseudo-device
IOC_PDEV_SET_BUF_PTRS	Set request firstByte and read-ahead lastByte
IOC_PDEV_REPLY	Give return code and the address of the results
IOC_PDEV_READY	Indicate the pseudo-device is ready for I/O

Table 7-3. The server uses these **ioctl** calls to complete its half of the request-response protocol. The first two operations are invoked to set up the request buffer, and the remaining three are used when handling requests.

whenever it needs to know whether a connection is ready, such as during **read**, **write**, and **select** calls. Pseudo-devices were originally implemented this way. Unfortunately, it resulted in an enormous number of context switches into and out of server processes. The worst case was a client process issuing a select call on several pseudo-devices; most of the time most pseudo-devices were not ready, so the servers were invoked needlessly.

This problem was fixed by having the kernel maintain three bits of state information for each request-response connection, corresponding to the readable, writable, and exception masks for the **select** call. The pseudo-device server updates these bits each time it replies to a request, and it can also change them with the IOC_PDEV_READY **ioctl**. This mechanism allows the kernel to find out whether a pseudo-device is ready without contacting the server, and it resulted in a significant performance improvement for **select**. In addition, the server can return the EWOULDBLOCK return code from a READ or WRITE request; the kernel will take care of blocking the process (unless it has requested non-blocking I/O) and will reawaken the process and retry its request when the pseudo-device becomes ready again. Thus the pseudo-device server determines whether or not the device is ready, but the kernel handles the logistics of blocking and unblocking processes. With respect to this blocking protocol, pseudo-devices behave exactly like the objects implemented by the kernel.

7.4.4. Future Work

There are two additional aspects of the Sprite file system architecture that have not been extended (yet) for use with user-level server process: data caching and automatic recovery. Currently the kernel's data cache is only used with files, but it could be extended to cache data for a user-level server. The cache defines its own back-end interface to read and write cache blocks, and the existing read and write procedures for pseudo-device connections can be reused for this purpose. Additional server **ioctl** commands need to be defined so the server could invalidate blocks and force other blocks to be written out. The additional complexity to the kernel would not be that great because the cache already provides cache-control functions and a generic back-end to support its use for both local and remote files.

The recovery system described in Chapter 6 can also be extended to support user-level servers. Currently there is no recovery, so if a server process crashes then the I/O streams it serviced are closed. The recovery system is based on keeping redundant state, so to utilize the existing kernel mechanisms the user-level server has to register state with the kernel for each I/O stream it services. In the case of remote clients, the per-stream state should be propagated back to the remote client kernel for safe-keeping. The net result is that per-stream state is duplicated in the kernel (or kernels) involved with the pseudo-device. This will allow recovery either from a crashed server process or from the crash of the host running the server process. When the server process is restarted, it would first see a series of REOPEN requests that include the per-client state previously registered by the last server process. While this will not be feasible for all user-level servers, it would be quite straight-forward for the NFS pseudo-file-system. The per-stream state for the NFS pseudo-file-system server is a 32-byte NFS handle that identifies the file to the NFS file server, and this state does not change once the stream is opened.

7.5. Performance Review

This section presents some performance measurements of the implementation. First the request-response protocol is measured, including the effect of using write-behind. Then the performance of actual pseudo-devices (for the UDP protocol) and pseudo-file-systems (for NFS access) is examined.

7.5.1. Request-Response Performance

There are a number of contributions to the cost of the request-response protocol: system call overhead, context switching, copy costs, network communication, synchronization, and other software overhead. The measurements below compare the request-response protocol with UNIX TCP sockts and UNIX UDP sockets, and pipes under both Sprite and UNIX. The hardware used in the tests is a Sun-3/75 with 8 megabytes of main memory, and the UNIX is SunOS 3.2.

Each of the benchmarks uses some communication mechanism to switch back and forth between the client and the server process. Each communication exchange requires four kernel calls and two context switches. With pseudo-devices, the client makes one system call and gets blocked waiting for the server. The pseudo-device server makes three system calls to handle each request: one to read the request stream, one to reply, and one to update the firstByte pointer into the request buffer. (These last two calls could could be combined, but currently they are not.) With the other mechanisms the client

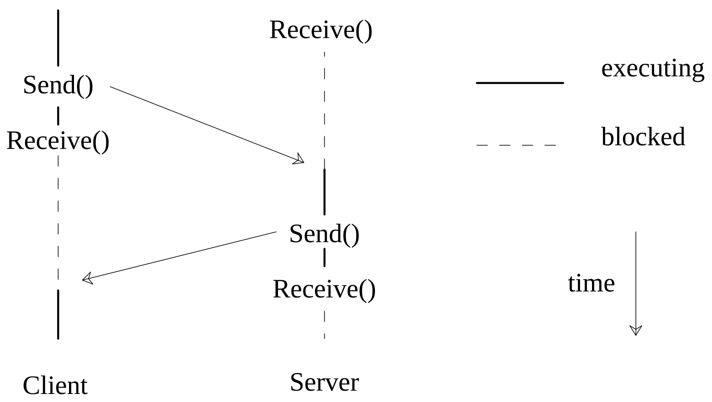


Figure 7-5. This shows the flow of control between two processes that exchange messages. Initially the server is waiting for a message from the client. The client sends the message and then blocks waiting for a reply. The client executes again after the server waits for the next request. Each benchmark has a similar structure, although different primitives are used for the Send and Receive operations shown here.

makes two system calls: one to make a request of the server and another to wait for a response. The server makes two system calls as well: one to respond to the client, and one more to wait for the next request. The flow of control is shown in Figure 7-5.

Table 7-4 presents the elapsed time for a round-trip between processes for each mechanism when sending little or no data. The measurements were made by timing the cost of several thousand round-trips and averaging the results. The measured time includes time spent in the user-level processes. The costs of communication via pipes (on UNIX and Sprite) and pseudo-device (Sprite only) are roughly the same in the local case, which suggests that overhead from context switching and the scheduler, which applies to all cases, is the dominant cost. In the second half of the table, the client and server processes are on different hosts so there are network costs. The network cost of using Sprite RPC and UNIX-to-UNIX UDP is about the same at these small transfer sizes. However, the Sprite RPC protocol provides reliable message exchanges while UDP does not. TCP, which does provide a reliable channel, is much more expensive than using remote pseudo-devices or UDP.

The difference between exchanging zero bytes and one byte using pseudo-devices highlights the memory mapping overhead incurred in the pseudo-device implementation. When the kernel puts a request into the server's buffer it is running on behalf of the client process. On the Sun hardware only one user process's address space is visible at a time, so it is necessary to map the server's buffer into the kernel's address space before

Process Communication Latency (microseconds)			
Benchmark	Bytes	Sprite	UNIX
PipeExchange	1	1910	2180
Pseudo-Device	0	2050	-
Pseudo-Device	1	2440	-
UDP socket	1	-	1940
TCP socket	100	-	5180
Remote Pdev	0	4260	-
Remote Pdev	1	5000	-
Remote UDP	1	-	4870
Remote TCP	100	-	7980

Table 7-4. The results of various benchmarks running on a Sun-3/75 workstation under Sprite and/or UNIX. Each benchmark involves two communicating processes: PipeExchange passes one byte between processes using pipes, Pseudo-Device does a null **ioctl** call on a pseudo-device, UDP exchanges 1 byte using a UNIX UDP datagram socket, and TCP exchanges 100 bytes using a UNIX TCP stream socket. The TCP test moves larger chunks of data because the TCP implementations delay small messages in an attempt to batch up several small messages into one network packet.

copying into it. Similarly, when the server returns reply data the client's buffer must be mapped in. The mapping is done twice each iteration because data is sent both directions, and obvious optimizations (i.e. caching the mappings), have not been implemented.

Figure 7-6 shows the performance of the various mechanisms as the amount of data varies. Data is transferred in both directions in the tests, and the slope of each line gives the per-byte handling cost. The graphs for UDP and TCP are non-linear due to the *mbuf* buffering scheme used in UNIX; messages are composed of chains of buffers, either 112 bytes or 1024 bytes. Messages that are not multiples of 1024 bytes suffer a performance hit because they can involve long buffer chains.

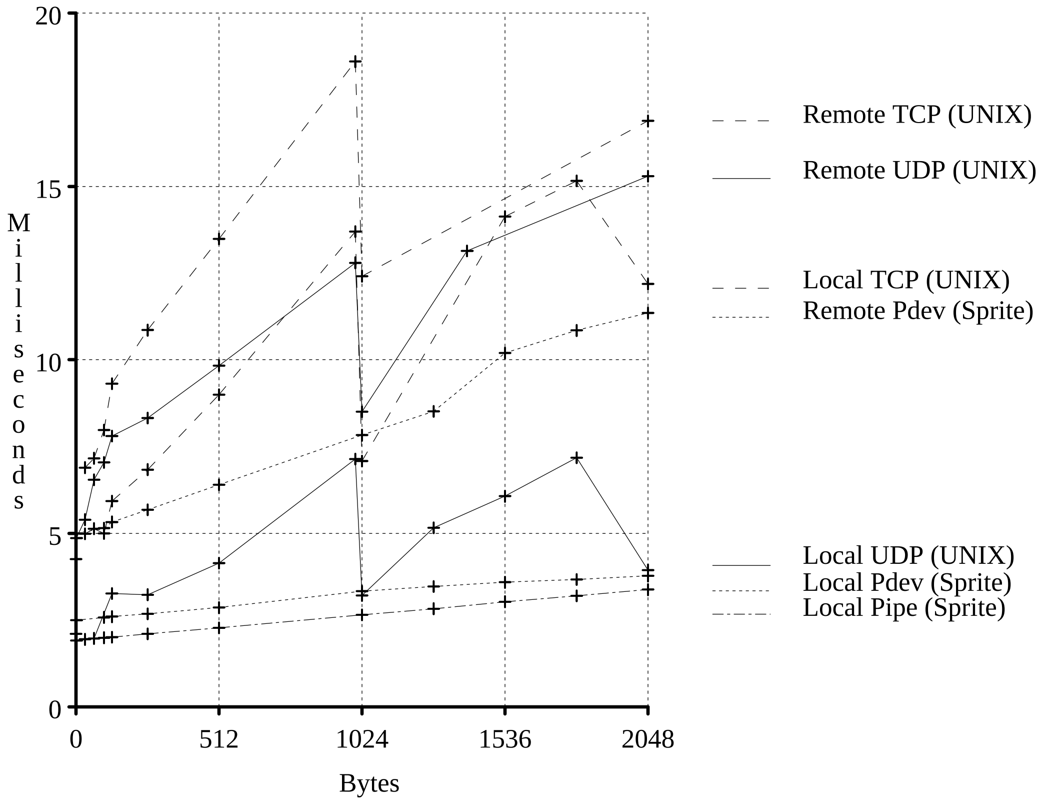


Figure 7-6. Elapsed time per exchange vs. bytes transferred, local and remote, with different communication mechanisms: Pseudo-devices, Sprite pipes (local only), and UNIX UDP and TCP sockets. The bytes were transferred in both directions. The shape of the UDP and TCP lines is due to the buffering scheme used in UNIX. The buffers are optimized for packets that are multiples of 1 Kbyte, and chains of 112-byte buffers are used with packets of intermediate size.

The Sprite mechanisms have nearly constant per-byte costs. The unrolled byte copy routine used by the kernel takes about 200 microseconds per kbyte. Data is copied four times using pipes because there is an intermediate kernel buffer. The measured cost is about 800 microseconds per kbyte. Data is copied twice using pseudo-devices, and we expect a per-kbyte cost of 400 microseconds. Close examination of Figure 7-6 reveals that the pseudo-device cost is close to 400 microseconds/kbyte with transfers between 1-kbyte and 2-kbytes, while it is slightly larger than this a smaller buffer sizes. The overall difference is slight, however, so this anomalies may be due to experimental error.

In the remote case, the pseudo-device implementation uses one kernel-to-kernel RPC to forward the client's operation to the server's host. The RPC adds about 2.4 msec to the base cost when no data is transferred, and about 4.3 msec when 1 kbyte is transferred in both directions. There is a jump in the remote pseudo-device line in Figure 5 between 1280 and 1536 bytes when an additional ethernet packet is needed to send the data.

The effects of write-behind buffering can be seen by comparing the costs of writing a pseudo-device with and without write-behind. The results in Table 7-5 show a 60% reduction in elapsed times for small writes. This speed-up is due to fewer context switches between the processes, and because the server makes one less system call per iteration because it does not return an explicit reply. The table also gives the optimal number of context switches possible, and the actual number of context switches taken during 1000 iterations. The optimal number of switches is a function of the size of each request and the size of the request buffer (2048 bytes in this case). Preemptive scheduling causes extra context switches. The server has a 2048-byte request buffer and there is a 40-byte header on requests, so, for example, 28 write messages each with 32 bytes of

Pseudo-Device Write vs. Write-behind				
(Bytes vs. Microseconds & Context Switches)				
Size	Write	Write-Behind	Ctx Swtch	Opt Swtch
32	2330	910	40	36
64	2370	940	63	53
128	2400	1000	100	84
256	2450	1120	178	167
512	2590	1420	382	334
1024	3030	2660	2000	1000

Table 7-5. The elapsed time in microseconds for a write call with and without write-behind, and the number of context switches taken during 1000 iterations of the write-behind run vs. the optimal number of switches. The write-behind times reflect a smaller number of context switches because of write-behind. The optimal number of switches is not obtained because the scheduler preempts the client before it completely fills the request buffer.

data will fit into the request buffer, but only one write message with 1024 bytes of data will fit. A scheduling anomaly also shows up at 1024 bytes; the kernel's synchronization primitives cause the client process to be scheduled too soon, so there are twice as many context switches as expected.

7.5.2. UDP Performance

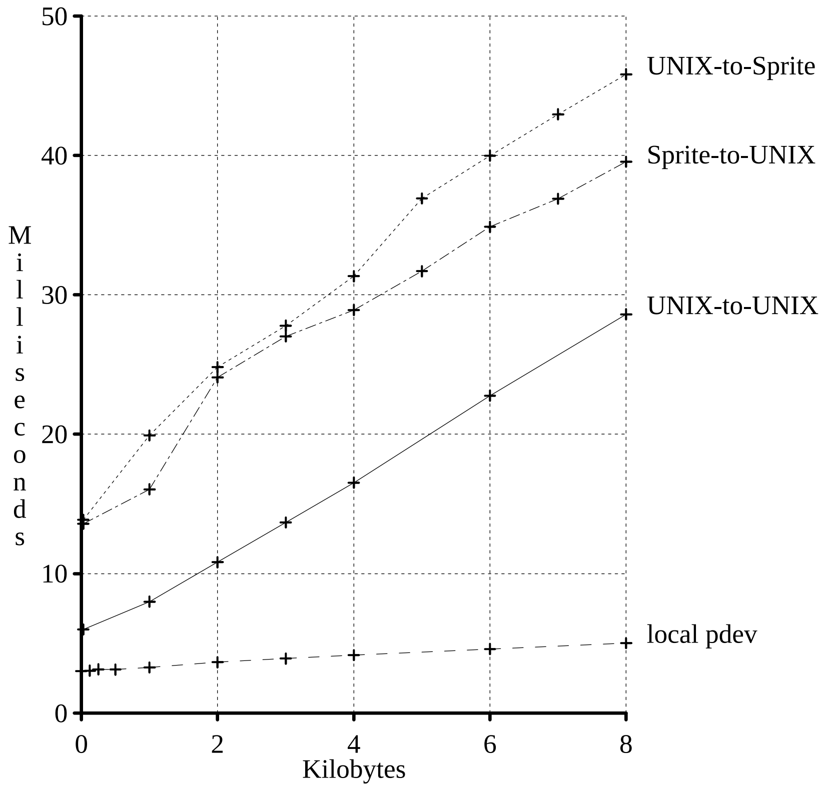


Figure 7-7. Timing of the UDP protocol. The receiver is always a UNIX process to model the use of UDP to communicate with the UNIX NFS server. Each Sprite-to-UNIX packet exchange requires two request-response transactions with the Sprite UDP server. The cost of accessing the UDP service via the pseudo-device request-response protocol is given by the line labeled ''local pdev''. The small slope of this line indicates that copy costs are not that significant but process scheduling and context switching have a large impact on performance. Note that message sizes are multiples of 1024 bytes, which are the best cases for the UNIX UDP implementation; the non-linear effects from odd buffer sizes, which were shown in Figure 7-6, are not present here.

The performance penalty of moving a service out of the kernel is demonstrated by the cost of the UDP datagram protocol, which is implemented in Sprite using a pseudo-device. The cost to send data via a UDP packet and receive a one-byte acknowledgment packet is plotted in Figure 7-7. The UNIX-to-UNIX time is the cost of a kernel-based UDP protocol. The Sprite-to-UNIX and UNIX-to-Sprite times are slower because of the user-level pseudo-device server on the Sprite side. The cost of sending data to the pseudo-device server is plotted as the line labeled ‘‘local pdev’’. The difference between the Sprite-to-UNIX and UNIX-to-UNIX UDP times comes from paying this cost twice, once for each packet exchanged. There is also an additional penalty when receiving a large UDP datagram in Sprite. UNIX can do IP fragment reassembly at interrupt time, while a full context switch is required to the server in Sprite for each fragment of the message. Output of large messages is not slowed in Sprite because a context switch is not suffered between output of each packet fragment.

7.5.3. NFS Performance

The performance of the NFS pseudo-file-system was measured with micro benchmarks that measure individual file system operations, and with a macro benchmark that measures the system-level cost of pseudo-file-system access. The cost of raw I/O operations through a pseudo-file-system is obviously going to be higher than the cost of I/O operations implemented by the kernel. Our NFS access involves two user-level servers for communication: one for the UDP protocol and one for the NFS protocol. However, when whole applications are run the effect of pseudo-file-system access is less pronounced.

The tests were run on Sun-3 workstations that run at 16 MHz and have 8 to 16 Mbytes of main memory. The network is a 10 Mbit Ethernet. The file servers are equipped with 400-Mbyte Fujistu Eagle drives and Xylogics 450 controllers. The version of the Sun operating system is SunOS 3.2 on the NFS clients, and SunOS 3.4 on the NFS file servers.

The four cases tested are:

Sprite — A Sprite application process accessing a Sprite file server. File access is optimized using Sprite’s distributed caching system [Nelson88a].

UNIX-NFS — A UNIX application process accessing an NFS file server. ‘‘/tmp’’ is located on a virtual network disk (ND) that has better writing performance than NFS.

Sprite-NFS — A Sprite application accessing an NFS file server via a pseudo-file-system whose server process is on the same host as the application. A Sprite file server is used for executable files and for ‘‘/tmp’’.

Sprite-rmt-NFS — A Sprite application accessing NFS from a different host than the pseudo-file-system server’s host.

The raw I/O performance for Sprite files, NFS files, and NFS files accessed from Sprite is given in Table 7-6. In all cases the file is in the file server’s main memory

Read-Write Performance			
Read 1-Meg	UNIX-NFS	320 K/s	25.0 msec/8K
Read 1-Meg	Sprite	280 K/s	14.3 msec/4K
Read 1-Meg	Sprite-NFS	135 K/s	59.3 msec/8K
Read 1-Meg	Sprite-rmt-NFS	75 K/s	106.7 msec/8K
Write 1-Meg	UNIX-NFS	60 K/s	133.3 msec/8K
Write 1-Meg	Sprite	320 K/s	12.5 msec/4K
Write 1-Meg	Sprite-NFS	40 K/s	200.0 msec/8K
Write 1-Meg	Sprite-rmt-NFS	31 K/s	258.0 msec/8K

Table 7-6. I/O performance when reading and writing a remote file. The file is in the server's main-memory cache when reading. Sprite uses s 4-Kbyte block size for network transfers while NFS uses an 8-Kbyte block size. The write bandwidth is lower when accessing the NFS server because it writes its data through to disk while the Sprite file server implements delayed writes.

cache. Ordinarily Sprite caches native Sprite files in the client's main memory, too. For the read benchmark I flushed the client cache before the test. For the write benchmark I disabled the client cache. The native Sprite read bandwidth is lower than NFS read bandwidth because Sprite uses a smaller blocksize, 4K verses 8K. The native Sprite write bandwidth is an order of magnitude greater than NFS write bandwidth because NFS file servers write their data through to disk before responding, while Sprite servers respond as soon as the data is in their cache.

The system-level performance of the NFS pseudo-file-system was measured using the Andrew file system benchmark. This benchmark was developed at CMU by M. Satyanarayanan [Howard88]. It includes several file-system-intensive phases that copy files, examine the files a number of times, and compile the files into an executable pro-

Andrew Benchmark Performance		
Sprite	522 secs	0.69
UNIX-NFS	760 secs	1.0
Sprite-NFS	1008 secs	1.33
Sprite-remote-NFS	1074 secs	1.41

Table 7-7. The performance of the Andrew benchmark on different kinds of file systems. The elapsed time in seconds and the relative slowdown compared to the native NFS case are given.

gram. The results of running this benchmark are given in Table 7-7.[20] The NFS performance is 33-41% slower than using a native kernel implementation of NFS. However, the table really highlights the difference between the high-performance kernel implementation of the Sprite file system, the mediocre performance of native NFS, and the further cost of accessing this via a user-level server process.

7.6. Related Work

There are two ways to characterize the pseudo-file-system and pseudo-device mechanisms, and thus two ways to compare them against existing work. First, they are a means of extending the distributed file system with new functionality without modifying the kernel. The new services benefit from features of the file system like the name space, remote access, and blocking I/O, but they are free to implement non-file-like functions via the **ioctl** call. Second, these mechanisms provide a form of interprocess communication (IPC). They are oriented towards the client-server model because one server has connections to one or more clients. Of course, these two characteristics complement each other. The file system is organized around a client-server model, and the pseudo-device and pseudo-file-system mechanisms allow the server to be implemented at user-level instead of inside the kernel.

There are a number of systems that implement system services outside the kernel. In message passing systems, the kernel just implements processes, address spaces, and a message-passing or RPC protocol for IPC. Services like the file system are not implemented in the kernel in these systems. The V-system[Cheriton84], Mach[Accetta86], and Amoeba[Renesse89] are a few examples, and there are others [Bershad89] [Fitzgerald85] [Scott89]. Sprite, in contrast, takes a hybrid approach where the file system is implemented inside the kernel, but other services can be implemented at user-level. This approach gives the file system a performance advantage; there are no costs associated with message passing and context switching during regular file access. Furthermore, user-level services in Sprite are well integrated into the system so they benefit from existing kernel features like the distributed name space, and the integration is transparent so an application cannot tell if it is accessing a kernel-resident or a user-level service.

Another approach to adding system services is to implement them in a run-time library. As with the message-passing systems, this aproach has the attraction that it is not necessary to modify the kernel to add new functions. Often these systems support dynamic linking and/or shared libraries so that applications do not have to incorporate every library procedure into their address space. Some early distributed file systems were implemented in the run-time library, such as UNIX United[Brownbridge82]. The Apollo DOMAIN file system implements an extensible I/O stream facility in its shared

[20] The version we used here has been modified to eliminate machine dependencies, so the results are not directly comparable with those reported in [Howard88] and [Nelson88a]. The compilation phase, in particular, is more CPU-intensive in the version used here.

run-time library [Rees86]. The primary advantage that Sprite has over library approach is that network communication is handled efficiently by the Sprite kernel, as opposed to being implemented the library. In the description of the DOMAIN facility, for example, there is no mention of network access to a library-level stream manager, which suggests that it would have to be coded explicitly. In Sprite, network access to a user-level server reuses the kernel network communication facilities.

With respect to integrating user-level servers into the file system, there are two systems that achieve similar functionality as the Sprite pseudo-device and pseudo-file-system mechanisms: the UNIX Version Eight stream facility [Ritchie84] and Bershad's Watchdog mechanism[Bershad88]. The UNIX stream facility provides byte stream connections between processes, but it can be extended to allow emulation of an I/O device by a user-level process [Presotto85]. One extension converts **ioctl** calls on the byte-stream into special messages that appear in the byte stream. Another extension lets the server ''mount'' a stream on a directory. In this case naming operations that encounter this directory are converted into messages on the byte-stream. The server process interprets the remaining pathnam, and it returns an open I/O stream in response to an open request.

There are two main differences between the streams facility and the Sprite pseudo-device and pseudo-file-system mechanisms. First, with the streams facility the server has a single byte-stream connection that it must multiplex among clients. In Sprite, there are distinct request streams for each client, and the request-response nature of the communication is explicit. The Sprite user-level server is simpler because it does not have to juggle several service requests over a single byte-stream. The second difference is that the Sprite file system provides a distributed name space and transparent remote access. A single instance of a pseudo-device or pseudo-file-system server can be shared by all hosts in the Sprite network. In contrast, each streams server is private to the UNIX host, although it is possible to have server-to-server communication via network protocols.

The watchdog facility proposed by Bershad and Pinkerton [Bershad88] provides a different way to extend the UNIX file system. A ''watchdog'' process can attach itself to a file or directory and take over some, or all, of the operations on the file. The watchdog process is an un-privileged user process, but the interface is implemented in the kernel so the watchdog's existence is transparent. Watchdogs may either wait around for guarded files to be opened, or they are created dynamically at open-time by a master watchdog process. There are two advantages of the Sprite user-level server mechanisms over Watchdogs. First, the Sprite read-write interface can be asynchronous to reduce context switching costs. Second, network access to the user-level server is provided by the Sprite kernel, while Watchdogs were implemented in a stand-alone UNIX system. One advantage of the Watchdog facility is that the server can choose which operations to implement, and defer the rest to the kernel. The granularity is coarser in Sprite, with the user-level server implementing either the I/O interface or the naming interface, or both.

Thus, the main advantages of the Sprite pseudo-device and pseudo-file-system mechanisms have over other systems are the features provided by the Sprite distributed file system architecture. This include the distributed name space, transparent remote access, and blocking I/O. Also, the existing kernel mechanisms for data caching and automatic error recovery can be extended for use with user-level services, although I have not had the time to implement these extensions.

7.7. Conclusion

Pseudo-file-systems and pseudo-devices are an extension of the architecture already present in Sprite to support its distributed file system. Pseudo-file-systems are treated as another domain type that is automatically integrated into the name space by the prefix table mechanism. Remote access is handled by the kernel with the same mechanisms used to access remote Sprite servers. Pseudo-devices are named and protected just like other files, while pseudo-file-systems define their own naming and protection systems. By making the service appear as part of the file system the existing open-close-read-write interface is retained. Byte stream communication is via **read** and **write**, and read-ahead and write-behind can be used for asynchronous communication. **Ioctl** is available for operations specific to the service, and can be used as the transport mechanism for a user-level RPC system. The standard interface means that the implementation of a service could be moved into the kernel for better performance without having to change any clients.

Our performance measurements show a distinct penalty for user-level implementation. We knew in advance this would be true, but we have found the performance of our user-level services to be acceptable. However, an important lesson from this work is that user-level servers are not the best solution for all problems. Much of the recent research in operating systems has focused on means of pushing services, including the file system, out of the kernel. However, because the file system is so heavily used, it makes sense to optimize its performance with a kernel-based implementation. On the other hand, it is also important to have a means to move functionality out of the kernel so that the kernel does not get bloated with extra features. The pseudo-file-system and pseudo-device mechanisms provide a convenient framework for user-level services, and they benefit from mechanisms already present in the Sprite kernel to support the distributed file system.

CHAPTER 8

Caching System Measurements

8.1. Introduction

This chapter motivates the need to have a stateful system with a review of the Sprite caching system[Nelson88b] and some measurements of its behavior on our network. In Sprite, both clients and servers cache file data in their main memories to optimize I/O operations. The main memory caches improve the performance of reading data because network and disk accesses are eliminated if recently-used data is found in the cache. Write performance is also improved because data is allowed to age in the cache before being written back to the server. Data is written to the server or the disk in the background so applications do not have to wait for the relatively slow disks. Some write operations are eliminated because data is deleted or overwritten before it is written back. With this caching system the performance of file I/O can scale with CPU speeds; it is not always limited by disk or network performance.

This chapter provides a follow-on study to Nelson's thesis [Nelson88b]. Nelson studied the effects of different writing policies on the Sprite caching system and the interactions of the caching system and the virtual memory system. He used a set of benchmarks to evaluate the system. The results presented here are based on statistics taken from the system as it is used for day-to-day work by a variety of users. The raw

Summary of Caching Measurements	
Files requiring consistency callbacks	1% opens
Files uncachable because of sharing	8% opens
Dirty files re-read by a client	13% opens
Files concurrently read shared	36% opens
Average client cache sizes	17%-35% memory
Average server cache sizes	25%-61% memory
Average client read miss ratios	36% bytes
Average client write traffic ratios	53% bytes

Table 8-1. Summary of results. The first half of the table contains cache consistency related figures. The second half contains cache effectiveness figures.

data is in the form of about 450 different counters that are maintained in the kernel and periodically sampled. File servers were sampled hourly, and clients were sampled 6 times each day. From this data it is possible to compute I/O rates, cache hit ratios, the number of RPCs serviced, and other statistics. The main results presented in this chapter are summarized in Table 8-1. There are too many statistics to fit them all in this chapter. Additional measurements are presented in Appendix B, and the raw data for the period from July through December, 1989 is available to other researchers. The study period is interesting because the system more than doubled in size over the study period with the addition of new high-performance workstations. Table 8-2 lists the characteristics of the hosts that made up the Sprite network during the study period.

The remainder of this chapter is organized as follows. Section 8.2 reviews the Sprite caching system and the algorithm used to maintain consistency of client caches. Section 8.3 presents results on the cache consistency overhead. Section 8.4 presents measurements of the effectiveness of the caching system during normal system activity. Section 8.5 shows how variable-sized caches dynamically adapt to clients and servers of different memory sizes. Section 8.6 concludes the chapter.

8.2. The Sprite Caching System

This section reviews the Sprite caching system originally described in [Nelson88a]. The important properties of Sprite's caching system are: 1) diskless clients of the file system use their main memories to cache data, 2) clients use a delayed-writing policy so that temporary data does not have to be written to the server, and 3) the servers guarantee that clients always get data that is consistent with activity by other clients, regardless of how files are being shared throughout the network. Servers also cache data in their main memory and use delayed writes, and the implementation of the client and server caches is basically the same.

8.2.1. The Sprite Cache Consistency Scheme

The Sprite file servers must solve a cache consistency problem: the same data may be cached at many locations, and it is important that these versions remain consistent with each other. In order to provide a consistent view of file data, the file servers keep state about how their files are being cached by clients, and they issue cache control messages to clients so that clients always get the most up-to-date file system data. This approach is a centralized approach to cache consistency, with the burden of maintaining consistency placed on the file servers. Nelson describes the two cases that servers have to address, *concurrent write sharing* and *sequential write sharing* [Nelson88b]. These cases are illustrated in Figure 8-1.

Sequential write sharing occurs when Client A reads or writes a file that was previously written by Client B. In this case, any cached data for the file in Client A's cache from an earlier version is invalid. A version number is kept for each file to detect this situation. Each time a file is opened for writing the file server increments the version

Characteristics of Sprite Hosts					
Name	Model	MIPS	Mbytes	User	Birthday
Mint	Sun 3/180	2	16	FileServer	Jun 3 '87
Oregano	Sun 3/140	2	16	FileServer	Jun 10 '88
Allspice	Sun 4/280	9	128	FileServer	Jul 17 '89
Assault	DS3100	13	24	FileServer	Sep 13 '89
Sloth	Sun 3/75	2	8	Spriter	Sep 17 '87
Mace	Sun 3/75	2	8	Faculty	Oct 11'87
Murder	Sun 3/60	3	16	Spriter	Oct 20 '87
Paprika	Sun 3/75	2	12	Spriter	Oct 20 '87
Sage	Sun 3/75	2	12	Spriter	Oct 20 '87
Thyme	Sun 3/75	2	16	Spriter	Oct 21 '87
Nutmeg	Sun 3/75	2	8	Spriter	Dec 13 '87
Fenugreek	Sun 3/75	2	12	Spriter	Jun 15 '88
Basil	Sun 3/75	2	8	GradStudent	Oct 5 '88
Mustard	Sun 3/75	2	8	GradStudent	May 11 '89
Sassafras	Sun 3/75	2	8	GradStudent	Jun 9 '89
Lust	SPUR MP3	6	32	Testing	Apr 23 '89
Anise	Sun 4/260	9	32	Testing	Apr 10 '89
Jaywalk	Sun 4c	12.5	12	Testing	Aug 3 '89
Covet	Sun 4c	12.5	24	GradStudent	Sep 29 '89
Burble	Sun 4c	12.5	12	GradStudent	Oct 4 '89
Kvetching	DS3100	13	24	Spriter	Jul 14 '89
Cardamom	DS3100	13	24	Faculty	Jul 26 '89
Pride	DS3100	13	24	Spriter	Jul 22 '89
Hijack	DS3100	13	24	Spriter	Aug 1 '89
Piracy	DS3100	13	24	Shared	Aug 3 '89
Pepper	DS3100	13	24	Shared	Aug 8 '89
Parsley	DS3100	13	24	Faculty	Aug 20 '89
Violence	DS3100	13	24	Secretary	Aug 22 '89
Piquante	DS3100	13	24	Testing	Aug 23 '89
Forgery	DS3100	13	24	GradStudent	Aug 24 '89
Subversion	DS3100	13	24	GradStudent	Aug 24 '89
Apathy	DS3100	13	24	GradStudent	Aug 26 '89
Gluttony	DS3100	13	24	GradStudent	Sep 13 '89
Clove	DS3100	13	24	GradStudent	Oct 11 '89
Garlic	DS3100	13	24	GradStudent	Oct 16 '89

Table 8-2. Characteristics of the hosts in the Sprite network. They are sorted by machine type and their ''Birthday,'' the first recorded day they ran Sprite. (None of the Sun2s used in the initial development of Sprite are listed here.) The MIPS rating for each type of host is an approximation based on the Dhrystone benchmark; it is given as a rough comparison. ''Mbytes'' is the size of main memory of each host. User classifications are also given; Spriters are members of the Sprite development team; GradStudents are students working on other projects; Testing are hosts used primarily for testing and debugging.

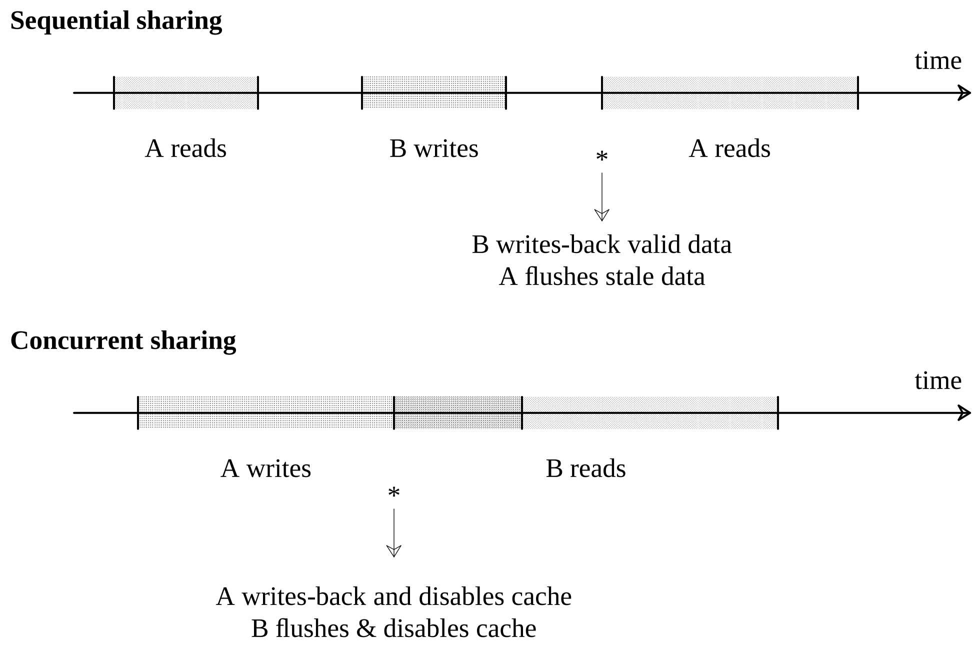


Figure 8-1. Concurrent and sequential write sharing of files introduces a cache consistency problem. In the case of sequential sharing the server issues a write-back command to the client caching the valid copy of the file, and the clients keep a version number to detect stale data. During concurrent sharing, the caching of the file is disabled on the clients and all I/O operations go through to the server's cache.

number. When a client opens a file it compares the version number returned from the file server with the version number of its cached file. If the client is opening for reading, a version number mis-match means it should discard (flush) its cached version. If the client is opening for writing there are two cases. One case is where the client has the previous version in its cache, and it is about to generate a new version. In this case the version number returned from the server will be one greater than the client's, and the client does not need to flush its cache. In the second case (server version number > client version + 1), however, the client does not have the most recent version of the file, so it flushes its old version. This flush ensures that partial updates of a cache block do not result in out-of-date data in the rest of the block. Finally, the delayed writing policy of the Sprite caching system means that the file server may not have the current version in its cache. In this case the file server issues a write-back command to the last writer (Client B) of the file before responding to the open request by Client A. The write-back command is suppressed if opening client is also the last writer.

Concurrent write sharing occurs when a file is open for writing on Client A during the same period of time the file is open for reading or writing on Client B. A simple approach to this situation is taken in Sprite. Caching of the shared file on the clients is disabled, and I/O operations go through to the cache on the file server. By disabling the remote caches, there is no danger of stale data being read on Client B while Client A is generating the valid version of the file. Studies in [Thompson87] and [Ousterhout85] indicate that concurrent write sharing is rare, so this simple approach should not degrade performance. Although these previous studies were based on trace data taken from timesharing systems, measurements presented below from our network show similar behavior in Sprite.

Not all distributed file systems provide the same level of consistency as Sprite. In Sprite, the caches on different hosts do not interfere with concurrent or sequential file sharing. Other systems often loosen their consistency guarantees concerning concurrent sharing. A typical approach is to write back data when a file is closed so that sequential sharers see valid data, but not to make any guarantees when a file is concurrently shared. This approach is used in NFS[NFS85] and AFS[Howard88].

A related problem with these systems (NFS and AFS) is that there is no provision for delayed writes on the clients. In these systems, processes are blocked at close-time while data is written back to the server. However, measurements presented below indicate that as much as 50% of the data generated by a client is deleted or overwritten shortly after being created. A delayed-write policy means that this short-lived data never even gets written to the server, saving network bandwidth and server CPU cycles.

Other file systems do provide consistent sharing during concurrent access, but they differ from Sprite in the way they support consistency. Apollo DOMAIN [Levine85] requires explicit locking calls, and it flushes data upon unlock to ensure consistency. Files are memory-mapped in DOMAIN, so the locks and flushes are done by the virtual memory system. LOCUS [Popek85] uses a token-passing scheme to pass ‘‘ownership’’ of a file among sharers. Burrow’s thesis describes a system which uses tokens at the byte level [Burrows88]. In a token-passing system a client must have the token for a file in order to access it, and the server is in charge of circulating the token among clients so that each receives a fair percentage of ownership time.

The relative merits of the Sprite cache consistency scheme and the token-passing schemes depend on the nature of concurrent write sharing. If files are open concurrently but actually used in a sequential manner, Sprite’s scheme would be worse. In this case, all Sprite client writes would go over the network, while with a token-based scheme a client could do several writes to its cache before giving up the token and writing back dirty data. If the files really are written concurrently, however, then Sprite’s scheme will require less message traffic. In this case a token-passing system requires two extra RPCs on each I/O, one from the client to the file server to request the token, and one from the file server to another client to retrieve the token. This assumes the worst case where different clients make successive I/O operations. (With any consistency scheme there is still a need for synchronization so that concurrent I/O operations do not interfere. Neither Sprite, MFS, or LOCUS piggy-back lock acquisition with token passing, so synchronization is not considered in this comparison.) Finally, if concurrent write sharing is rare, then performance is not as important as simplicity. Sprite’s scheme is simpler for two reasons. First, all conflict checking is done at open-time, while in a token-passing system conflict checking is done at every read and write. Second, there is no need to circulate a token among Sprite clients.

8.3. Sharing and Consistency Overhead Measurements

The amount of file sharing and the consistency-related traffic was measured by the file servers, and the results are given in Table 8-3. This data was collected by monitoring all open operations over a 20-day interval. The table lists values for each file server, and the combination of all servers together that represents the total file system traffic. The various cases in the table are explained below.

Non-File

This value indicates the number of directories, symbolic links, and swap files that were opened. These files are not cached on the clients. Swap files are not cached so that VM pages really leave the machine upon page-out. Directories and links are not cached in order to simplify the implementation. These could be cached, but the servers would have to issue cache control messages when directories were updated or links were changed.

Can't Cache

This value indicates the total percentage of files opened that were not cachable on the clients. The difference between this column and the ''Non-File'' column is the percentage of files that were concurrently write shared. Mint, the root server, experiences a lot of concurrent write sharing, 14%. Nearly all of the concurrent write sharing occurs on a shared load average database used to select hosts for process migration. A small amount is due to accidental sharing of append-only log files. Overall, 8% of the opens are of concurrently write-shared files.

Read Sharing

This value counts the number of files that were open for reading by more than one process at a time, either on the same or different clients. This case is relatively frequent; it happens in about 36% of the cases. Note that this value does not count files that lingered read-only in a client's cache while not used, which is not

File Sharing and Cache Consistency Actions							
	Num	Non-	Can't	Read	Last	Server Action	
Server	Opens	File	Cache	Sharing	Writer	Write-back	Invalidate
Mint	4536840	10%	24%	42%	15%	0.202%	0.045%
Oregano	1171180	34%	35%	37%	8%	0.073%	1.575%
Allspice	2863430	36%	36%	26%	14%	0.267%	0.010%
Assault	491080	49%	49%	29%	2%	0.308%	0.049%
Combined	9062527	23%	31%	36%	13%	0.212%	0.232%

Table 8-3. File sharing by clients and the associated server consistency actions over a 20-day period, October 29 through November 19, 1989. The results are given for each server, and then for the combination of all the servers together, which represents the total file system traffic.

measured because the servers do not need to track this to keep caches consistent.

Last Writer

This value counts the files that were generated in a client's cache and then re-read or re-written by the same client before the 30 second delayed write period expired. Each of the servers except Assault sees a significant amount of this case, about 13% overall. This percentage is quite close to the percentage of files open for writing (15%[21]), which suggests that most data is re-read or re-written shortly after it is generated. Mint, for example, has log files that can be repeatedly updated by the same client. Oregano serves ''/tmp'', and compiler and editor temporaries account for the reuse of dirty files. Allspice has the system source directories, and compiler output often gets re-read by the linker. Assault is too lightly loaded to experience much of this behavior.

Server Action

This value indicates how often the servers had to issue cache control messages. ''Write-back'' indicates how many times the last writer of a file was told to write its version back to the file server. ''Invalidate'' indicates how many clients had to stop caching a file they were actively using because it became shared after it was opened. (The number of cache invalidations because of sequential write-sharing was not measured. As described above, this is done by the clients by checking version numbers, so it is not measurable on the servers.) Write-backs happen in less than 1% of the cases, which indicates that sequential write-sharing (within the delay period) between clients is rather rare. Invalidations are also rare, except on Oregano as described below.

The invalidations on Oregano are due to a temporary file used by **pmake**, our parallel compilation tool that uses process migration. **pmake** generates a temporary file containing the commands to be executed on the remote host. Initially, this file is cached on the host running **pmake**. During migration it is open by both the parent (**pmake**) and the child (a shell that will execute the commands on the remote host). These processes share a read-write I/O stream that the parent used to write the file and the child will use to read it. When the child migrates to the remote host the file server detects this as a case of concurrent write sharing and issues a write-back and invalidate command to the host running **pmake**. If the parent closed the file before the migration this would appear as sequential write sharing and contribute to the ''Write-back'' column instead.

Trace data from UNIX time sharing systems indicates that concurrent write sharing is very rare, and so is sequential write sharing between different users[Thompson87]. The UNIX files that are write-shared are database-like files such as the user-login database and system log files. Thompson found that these UNIX files only accounted for 1% or 2% of the open traffic. In Sprite, concurrent write sharing occurs on the same kind of files (databases and logs), but it accounts for 8% of the opens. Virtually all of this sharing is due to a database of host load averages that is used to select idle hosts as targets of

---

[21] Refer to Appendix B, Table B-3.

migration. The database is updated once each minute by each host, so it is heavily shared. The process that updates a host's entry keeps the database file open all the time, thus forcing it to be uncachable on the clients. The heavy open traffic to this database occurs because virtually every compilation accesses the database in order to find an idle host on which to execute.

Note that the measurements in this section are in terms of files open, not bytes transferred. Thus they give an overall measure of file sharing, but not necessarily a measure of how effective caching will be. Measurements presented below indicate how much I/O traffic there is to uncachable files and what the cache hit ratios are.

8.4. Measured Effectiveness of Sprite File Caches

This section presents results on the I/O traffic of the clients and servers, and it shows how effective the caches are during normal system use. The results of the study back up our original hypothesis that performance is dependent on the amount of memory on the clients, the more the better[Nelson88a]. It is not sufficient to upgrade the CPU performance without also increasing memory size. Extra memory reduces the load on the server by reducing paging traffic and increasing the effectiveness of the file cache. If memory is too small on a client then its performance is dominated by paging traffic, and the benefits of a faster CPU are lost. The other result is that because the client caches are so effective, it takes a relatively large cache on the file servers to yield any further benefit, such as reducing disk traffic.

8.4.1. Client I/O Traffic

This section presents the overall I/O traffic of the clients. Two metrics are given, I/O traffic from applications to the cache, and network traffic to the file servers. The network traffic includes all system effects, including paging and traffic to uncachable files. It represents the total load on the server from the client. (More detailed breakdowns of the traffic are given in the next sub-section.) Results for individual client workstations are given. The I/O traffic of a particular client is a function of its use. In our network the Sun3s place the most load on the servers, even though they are the slowest. This is because the Sun3s are the principal machines used for Sprite development, which includes large compilation jobs and debugging sessions. The other hosts are used more by the newer Sprite users, who apparently place less load on the system.

Table 8-4 gives the cache size and I/O rates for the Sun3 clients. The average, standard deviation, and maximum observed values are given for the megabytes of cache, bytes/second transferred to and from the cache, and bytes/second transferred to and from the server. The data was obtained by recording the total amount of data transferred by a client every three hours. The rates were computed over 3-hour intervals. The standard deviation gives a variance over different times of day. The maximum value is the largest rate observed in all the 3-hour intervals. The very high maximum rates (e.g. 100 Kbytes/sec on Sassafras) result from large simulation jobs that page constantly for long

Sun3 Client I/O Traffic (Bytes/Second)									
	Cache Size			Cache Traffic			Network Traffic		
Host	Meg	dev	max	bytes/s	dev	max	bytes/s	dev	max
8 Meg									
Basil	0.98	0.80	3.88	323	1027	8496	382	1255	15476
write				149	381	3916	195	556	5545
Mace	1.33	0.91	4.88	742	1773	22222	581	1595	15643
write				275	512	4343	277	596	4441
Mustard	1.75	1.02	4.41	841	1894	11250	551	2063	29487
write				323	729	6364	333	1433	24254
Nutmeg	1.45	1.00	4.59	427	1770	20121	389	3943	57170
write				147	416	4198	165	695	8869
Sassafras	1.45	1.09	4.48	785	1601	14408	1271	6785	104469
write				214	526	5842	656	4658	72701
Sloth	1.12	0.94	4.31	929	2519	25140	1096	4126	75486
write				360	1315	17582	575	2006	32038
average	1.35	0.96	4.42	660	1764	16940	689	3294	49622
write				243	647	7041	352	1657	24641
12 Meg									
Fenugreek	3.84	1.81	8.23	904	2248	15198	628	2256	32142
write				311	751	7141	322	922	8786
Paprika	2.46	1.57	7.01	740	1806	13673	833	2866	32487
write				262	570	4560	295	746	6476
Sage	3.26	1.89	8.38	804	2873	66661	698	2722	56987
write				302	1962	79765	355	1340	42117
average	3.19	1.76	7.88	819	2309	31844	715	2615	40539
write				293	1094	30489	326	1003	19126
16 Meg									
Murder	5.79	2.96	12.59	6528	10736	73579	5863	10580	72507
write				572	1574	18699	564	1527	15544
Thyme	5.23	2.77	11.84	1128	2979	29651	744	2064	30169
write				351	1533	29700	383	1567	29857
average	5.51	2.86	12.22	3833	6857	51615	3309	6322	51338
write				462	1554	24200	473	1547	22700

Table 8-4. Cache sizes and I/O traffic on the Sun3 clients, which are listed according to their memory size. The averages for each memory size are also given. Read traffic is on the first row for each host; write traffic is on the second row. These numbers were obtained by recording the bytes transferred by each client every three hours and computing the rate over that interval. The standard deviations give the variability of the rates over different 3 hour intervals, including intervals of relative inactivity. The ‘‘Cache Traffic’’ is the rate that bytes are transferred to and from the cache. The ‘‘Network Traffic’’ is the rate that bytes are transferred to and from the server over the network. Network traffic includes paging traffic, accesses to uncachable files, and traffic from cache misses.

periods of time. The clients average about 1100 bytes/sec in combined read-write traffic to their cache, and about the same amount of traffic over the network. While the cache eliminates some network accesses, paging traffic adds additional network accesses. (A more detailed breakdown of the remote I/O traffic is given in the next sub-section.) The clients with larger memories have higher data rates, although this difference is most likely because they are used by the more intensive Sprite developers. Also, Murder has the tape drive used for nightly dumps, so its I/O traffic is skewed by its nightly scans of the file system.

In comparison, the study of UNIX timesharing hosts by Ousterhout *et al* [Ousterhout85] found per-user I/O rates of 300-600 bytes/sec when averaged over 10 minute intervals, and rates of 1400 to 1800 bytes/sec when averaged over 10 second intervals. These are rates for active users only, and they do not include paging traffic. The average rates obtained for Sprite clients, about 1100 bytes/sec, include periods of inactivity. The I/O rates of an active Sprite client are better estimated by the peak rates given in Table 8-4. The peak I/O rates to the client caches range from 12000 bytes/sec to over 60000 bytes/sec, averaged over three hours. These large peak I/O demands result from large jobs that page heavily for long periods of time.

Table 8-5 gives the I/O traffic on the DECstation clients. Note that the I/O rates to the cache of the DECstations are not much different than on the Sun3s, even though the DECstation CPU is six times faster. The highest average rates on the Sun3 clients (i.e., 1479 bytes/sec on Thyme) are higher than the highest average rates on the DECstation clients (i.e., 1232 bytes/sec on Gluttony). This similarity suggests that the nature of the user effects the long term I/O rates to the cache, and the speed of the workstation just increases the burstiness. (The 3-hour sampling granularity of these statistics precludes measurement of short term rates.) The main difference between the DECstation and the Sun3 clients is that there is less network traffic generated by the DECstations. Each DECstation has 24 Mbytes of physical memory, while the Sun3s only have 8, 12, or 16 Mbytes. The extra memory on the DECstations reduces paging activity and it allows for larger file caches.

8.4.2. Remote I/O Traffic

The I/O traffic presented above includes remote traffic that is due to uncachable data, such as directories and the shared migration database, and it includes traffic due to page faults. Improved statistics were taken over an 11-day period to determine the contributions to the network traffic. Results for read traffic are given in Table 8-6. Cache miss ratios range from 17% to 65%, and the overall read miss rate is about 36%. (This average does not include Murder, the client that does nightly tape backups.) Perhaps the most significant result, however, is that cache misses only account for 29% of the remote data being read. The rest is due to page faults (38%) and uncachable data (33%). The page faults are mainly to read-only code files (23%) as opposed to the swap files used for dirty pages (15%). Thus the caches are effective in reducing network traffic, but there is still a considerable load from paging traffic and uncachable data.

DECstation 3100 Client I/O Traffic (Bytes/Second)									
	Cache Size			Cache Traffic			Network Traffic		
Host	Meg	dev	max	bytes/s	dev	max	bytes/s	dev	max
Apathy	7.62	1.32	7.94	723	1810	13986	242	833	6272
write				323	1047	8755	210	452	3293
Cardamom	6.61	2.72	16.43	411	942	5387	136	379	3555
write				166	389	3974	98	229	1670
Clove	7.27	1.73	7.94	452	759	3000	179	544	3669
write				196	257	1267	113	165	811
Forgery	6.38	2.43	11.65	384	935	8942	174	460	4011
write				159	504	5841	121	446	5437
Gluttony	5.13	2.02	7.94	861	3456	30498	951	3485	32875
write				371	667	5241	205	332	2623
Hijack	6.06	2.29	10.25	466	1711	18747	644	2731	41967
write				233	534	4600	247	431	5981
Kvetching	4.76	2.00	7.94	886	1524	11638	895	2113	22260
write				247	584	5654	231	508	3505
Parsley	6.88	2.13	8.94	471	971	7506	211	712	7767
write				213	532	5240	121	229	1667
Pepper	6.99	2.54	13.93	671	2242	18808	285	1640	16792
write				339	1630	16812	201	1232	16478
Pride	4.57	3.05	13.45	505	2252	19031	435	5618	42357
write				210	612	5624	223	1835	25106
Subversion	6.99	2.02	8.94	304	850	5856	105	531	6437
write				138	339	2678	70	166	1330
Violence	6.48	2.95	8.94	325	1649	18396	132	3732	51252
write				144	321	2461	100	260	2212
average	6.31	2.27	10.36	512	1592	13483	356	1898	19935
write				214	618	5679	159	524	5843

Table 8-5. Cache sizes and read traffic on the DECstation clients, all of which have 24 Mbytes of main memory. Read traffic is on the first row for each host; write traffic is on the second row. These numbers were obtained by recording the bytes transferred by each client every three hours and computing the rate over that interval. The standard deviations give the variability of the rates over different 3 hour intervals, including intervals of relative inactivity. The ‘‘Cache Traffic’’ is the rate that bytes are transferred to and from the cache. The ‘‘Network Traffic’’ is the rate that bytes are transferred to and from the server over the network. Network traffic includes paging traffic, accesses to uncachable files, and traffic from cache misses.

Remote Read Traffic								
Host	Cache	Read	Miss	Remote	Miss	Code	Data	Uncache
Kvetching	3.21	533.23	16.84%	879.94	10.20%	12.96%	24.07%	52.61%
Fenugreek	5.89	394.46	18.63%	118.70	61.92%	27.68%	4.73%	5.51%
Parsley	10.58	170.08	22.10%	67.60	55.59%	30.42%	3.39%	9.99%
Piracy	9.89	176.19	25.45%	71.75	62.48%	30.80%	1.32%	5.33%
Paprika	2.79	225.23	27.22%	578.91	10.59%	12.81%	1.79%	74.72%
Mustard	3.86	352.80	30.96%	245.64	44.47%	30.30%	19.64%	5.31%
Thyme	5.21	368.75	41.14%	642.34	23.62%	12.41%	3.44%	60.30%
Mace	1.13	314.15	41.93%	338.31	38.94%	41.27%	16.42%	3.05%
Burble	0.91	18.39	44.55%	41.63	19.68%	62.17%	7.47%	10.38%
Sassafras	1.95	511.76	48.61%	463.85	53.63%	24.85%	17.11%	4.12%
Sage	2.13	432.27	49.24%	550.76	38.64%	30.05%	28.48%	2.60%
Gluttony	6.47	126.59	49.27%	281.46	22.16%	15.81%	1.38%	60.58%
Sloth	1.09	219.48	64.67%	415.00	34.20%	38.61%	23.39%	3.36%
combined*	-	3843.36	35.74%	4695.88	29.25%	25.64%	11.96%	32.91%
Murder	4.84	1330.80	89.37%	1338.38	88.86%	5.89%	1.12%	4.02%

Table 8-6. Remote read traffic over an 11-day period. ‘‘Cache’’ is the average megabytes of cache. ‘‘Read’’ is megabytes read from the cache, and ‘‘Miss’’ is the cache miss traffic. ‘‘Remote’’ is the megabytes read from the file server, and ‘‘Miss’’ is the contribution from cache misses to this amount. ‘‘Code’’ is remote bytes read due to VM page faults on read-only code pages. ‘‘Data’’ is remote bytes read due to VM page faults on data pages. ‘‘Uncache’’ is uncachable bytes read, mainly directories and the migration database.
* Murder is eliminated from the combined rates to eliminate effects from its nightly backups.

The large read traffic to uncachable data stems from a single application program that a few users run in order to continuously display the number of hosts available for process migration. The program scans the complete load average database every 15 seconds, and the clients that run this program stand out clearly in Table 8-6. With most clients, uncachable data accounts for just a few percent of the network read traffic. However, running this application can cause uncachable data to account for as much as 60% of the network read traffic! It is important to emphasize that this application is not needed for normal system operation. If the clients that run this program are factored out, then the remote traffic due to uncachable data is only 4% of the server read traffic, with 44% from cache misses and 52% from page faults. This is a more reasonable view of the network traffic.

Table 8-7 gives the contributions from the cache (51%), paging (37%), and uncachable files (11%) to the network write traffic. (The remaining 0.63% stems from writes to remote devices and remote windows.) This table also gives a true picture of the cache write traffic ratio, which is the ratio of bytes written to the server to the bytes written to the cache. The client traffic ratios range from 27% to 72%, averaging about 53%. These

Remote Write Traffic							
Host	Cache	Write	Ratio	Remote	Cache	Swap	Uncache
burble	0.91	23.06	27.02%	23.29	26.76%	17.29%	55.95%
parsley	10.58	120.05	39.77%	66.87	71.40%	6.45%	22.03%
paprika	2.79	109.37	39.78%	80.89	53.79%	29.58%	16.52%
piracy	9.89	106.00	43.72%	57.09	81.18%	2.95%	15.55%
mustard.ds	3.86	181.29	43.91%	131.11	60.72%	24.74%	14.17%
sassafras	1.95	207.82	46.52%	235.98	40.97%	50.14%	7.72%
fenugreek	5.89	134.35	53.63%	103.22	69.80%	14.89%	15.15%
kvetching	3.21	103.92	57.37%	185.06	32.22%	61.24%	6.31%
mace	1.13	179.20	57.69%	172.36	59.98%	31.72%	8.23%
sage	2.13	171.91	57.76%	244.25	40.65%	51.78%	5.79%
thyme	5.21	162.49	62.29%	161.58	62.64%	26.36%	8.83%
sloth	1.09	114.13	66.05%	201.03	37.50%	54.04%	8.46%
gluttony	6.47	94.84	72.13%	99.69	68.63%	5.91%	25.37%
combined	-	1708.44	52.65%	1762.42	51.04%	36.98%	11.29%

Table 8-7. Remote write traffic over an 11-day period. ‘‘Cache’’ gives the average cache size, in Mbytes. ‘‘Write’’ gives the Mbytes written to the cache during the period. ‘‘Ratio’’ gives the percentage of the ‘‘Write’’ column that was written out of the cache back to the server. ‘‘Remote’’ gives the amount of data written to the server, in Mbytes. This is broken down into the contribution from cache write-backs (‘‘Cache’’), paging traffic (‘‘Swap’’), and uncachable files (‘‘Uncache’’).

measurements indicate that the 30 second aging period for dirty cache data is effective in trapping short-lived data in the cache; about half the data is never written back. However, paging can be significant for some of the clients. Sassafras, for example, is a compute server used for long-running simulations (and processing of the data presented here!), and its paging traffic accounts for about half of its network traffic. Write traffic to uncachable files is also significant, and the bulk of this comes from periodic updates of the migration database by each client.

8.4.3. Server I/O Traffic

This section presents the I/O traffic from the standpoint of the file servers. In the case of a file server it is interesting to compare the traffic to its cache to the traffic to its disks. Two metrics are given, the ‘‘File Traffic’’ and the ‘‘MetaData Traffic.’’ *Metadata* is data on the disk that describes a file and where it lives on disk. This includes the descriptor that stores the file’s attributes, and the index blocks used for the file map. The ‘‘File Traffic’’ represents I/O to file data blocks as opposed to the metadata information. The combination of file traffic and metadata traffic gives the total disk traffic for the file server.

Figure 8-2 has the server I/O traffic for a combination of all servers averaged over a 6-month study period. The graphs indicate that the server caches are effective for reads,

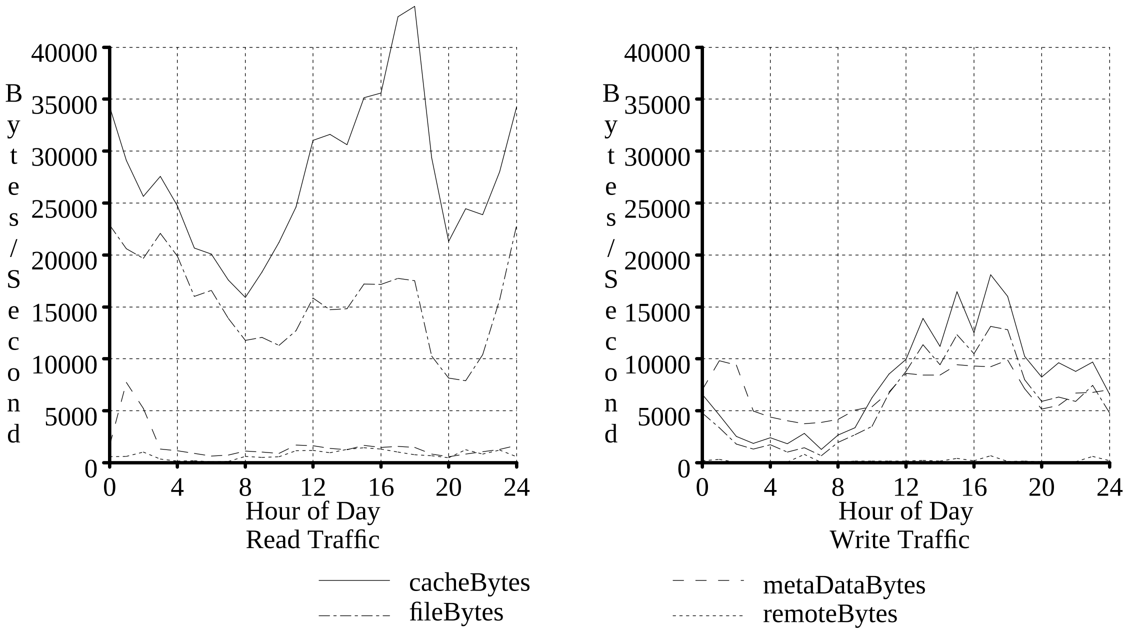


Figure 8-2. Server I/O traffic averaged from July 8 to December 22, 1989. The left-hand graph has read traffic and the write-hand graph has write traffic. The total disk traffic is the sum of ‘‘fileBytes’’ and ‘‘metaDataBytes’’.

but less effective for writes. Client caches trap out most of the short-lived data so most data written to a server ends up being written to disk. The write graph highlights the large amount of write traffic to metadata. File descriptors have to be updated with access and modify times, so just reading a file ultimately causes its descriptor to be written to disk. This effect has been noted by Hagmann[Hagmann87], who converted the CEDAR file system to log metadata changes, which reduced metadata traffic considerably.

Table 8-8 summarizes the results of a 20-day study period, from October 29 through November 19, 1989. This study was made after a bug was fixed that prevented continuously updated files from being written through to disk. With this bug present, Mint’s traffic ratio was about 50% for file data, while the other servers had write traffic ratios of 70% to 80% or more. Mint’s low traffic ratio prompted a search for a bug in the cache write-back code, and this 20-day study was made after it was fixed. Additional per-server measurements of I/O traffic are given in Appendix B, along with the detailed results of the 20-day study period.

The cache on Mint, the root server, is effective in eliminating reads (40% misses in the 20-day study), but not so good at eliminating writes (94% traffic ratio in the 20-day study). Its read hits occur on frequently-used program images and the load average database. About half the write traffic to Mint is to the load average database, which is

Server Traffic Ratios				
Host	Read	ReadAll	Write	WriteAll
Mint	40%	43%	94%	542%
Allspice	52%	57%	74%	107%
Oregano	72%	83%	86%	225%
Assault	82%	92%	55%	121%
Combined	54%	59%	77%	179%

Table 8-8. Summary of the results of a 20-day study of server I/O traffic. The ‘‘Read’’ and ‘‘Write’’ columns give the percentage of cache data that was transferred to and from disk. A low percentage means that the caches were effective in eliminating disk traffic. The ‘‘ReadAll’’ and ‘‘WriteAll’’ columns include traffic to metadata so they give the total disk traffic ratio. A more detailed breakdown of these results is given in Appendix B.

updated by each client once per minute. The metadata write traffic on Mint is quite high due to these frequently modified files. Furthermore, the 128-byte descriptors are written 32 at time in 4K blocks so there is extra traffic from unmodified descriptors.

Allspice and Oregano are directly comparable because they store the same type of files (many files were shifted from Oregano to Allspice during the 20-day study period). Allspice’s cache is about 10 times the size of Oregano’s and it is clearly more effective. This is to be expected because the server’s cache is a second-level cache, with the clients’ caches being the first level. The server’s caches have to be much larger than the client’s caches because the locality of references to their cache is not as good.

It is also interesting to see how the caches skew the disk traffic towards writes. During the 20-day study period, the traffic to the server caches was about 22% writes. The traffic to the server disks was 26% metadata writes and 20% data writes. If the metadata traffic is discounted as an artifact, then the data writes accounted for 40% of the disk traffic. The skew towards writes at the disk level should continue as the server caches get larger and more effective at trapping reads.

The server caches also have to be large enough in relation to the amount of disk space the server has. Measurements by Burrows [Burrows88], for example, indicate that 75% of the file system data is not accessed in over a week, and only 10% of it is modified in that time. Measurements of Sprite indicate that over 90% of the file system data has not been read over a day, and over 50% has not been read in over a month. These results hint that the active ‘‘working set’’ of a file system is probably much smaller than the amount of disk storage, and that a server cache that is a few percent of its disk space could cache a reasonable file system working set. In our system Mint has one 300-Mbyte disk and a 9-Mbyte cache, which is a ratio of cache to disk of 3%. Allspice has 2.4 Gbytes of disk storage and an 80-Mbyte cache, which is a ratio of about 3.3%. These two servers have rather effective caches. Oregano has four 300-Mbyte disks and about the same cache size as Mint, which is a ratio of 0.75%, and its cache is much less effective. Assault has 600 Mbytes of disk, and about 8 Mbytes of cache, for a ratio of 1.3%. It is a lightly used server, however, and half its disk space is rarely used at all. If the second disk is ignored then it also has a cache/disk ratio of about 3%, and its cache is also effective. Of course, it is dangerous to generalize from these few data points, but it seems clear that a server cache size that is a few percent of its disk space is better than a cache of 1% or less than the disk space.

8.5. Variable-Sized Caches

An important feature of Sprite caches is that they vary in size in order to make use of all available memory. Nelson[Nelson88b] explored ways of trading memory between the file system and the virtual memory system, which needs memory to run user programs. The basic approach he developed was to compare LRU times (estimated ages) between the oldest page in the FS cache and the oldest VM page and pick the oldest one for replacement. Nelson found that it was better to bias in favor of the VM system in order to reduce the page fault rate and provide a good interactive environment. The bias is achieved by adding a bias to the LRU time of the VM system so that its pages appear to be referenced more recently than they really were. We have chosen a bias against the file system of 20 minutes. Any VM page referenced within the last 20 minutes will never be replaced by a FS cache page. This policy is applied uniformly on all hosts, and it adapts naturally to both clients and servers. Servers use most of their memory for a file cache, while clients use most of their memory to run user programs.

8.5.1. Average Cache Sizes

Table 8-9 gives the average and maximum cache sizes as measured over the study period. The file servers are listed individually. The clients are grouped according to the amount of physical memory and processor type of the host, and the results are averaged. The adaptive nature of the cache sizes is evident when comparing clients and servers with the same memory size; the file servers devote more of their memory to the file cache. This difference is not achieved via any special cases in the implementation, but merely by the uniform application of the 20-minute bias against the file system described above.

The cache occupies a larger percentage of main memory as the memory size increases, indicating that the extra memory is being utilized more by the file cache than by the VM system. This trend is most evident on the Sun3 clients, of which there is a good population size, and in which the Sprite implementation is the most mature. Doubling the physical memory on a Sun3 client more than doubles the average cache size on the client; it increases from 17% to 34% of the physical memory.

The variability of the client caches is indicated by the standard deviation and the maximum observed values. The variability tends to increase as the memory gets larger, indicating that the cache is trading more memory with the VM system. The DECstations have lower variability because their cache was limited to about 8.7 meg during most of the study period. (This limitation is a software limit that has been recently lifted.)

Cache Size (Megabytes)							
Host	Mem	Average		Std Dev		Maximum	
Allspice*	128	67.27	52%	22.14	17%	78.13	61%
Assault	24	7.49	31%	4.55	19%	16.50	69%
Mint	16	9.03	56%	1.23	8%	11.80	74%
Oregano	16	8.59	54%	1.77	11%	12.06	75%
Sun3	8	1.35	17%	0.96	12%	4.42	55%
Sun3	12	3.19	27%	1.76	15%	7.88	66%
Sun3	16	5.51	34%	2.86	18%	12.22	76%
Sun4	12	2.09	17%	1.73	14%	6.87	57%
Sun4	24	6.00	25%	3.70	15%	13.43	56%
DS3100	24	6.31	26%	2.27	9%	10.36	43%

Table 8-9. Cache sizes as a function of main memory size and processor type, averaged over the 4-month study period. The average, standard deviation, and maximum values of the observed cache sizes are given. The sizes are megabytes and percentage of main memory size. The file servers are listed individually. The rest of the clients are averaged together based on CPU type and memory size.
* Allspice's cache was limited to at most 78.13 Mbytes.

The results from the servers show that Mint and Oregano can only devote a little over half their memory to their file cache. The reason they cannot use more is due to the increase in their kernel size as they accumulate more data structures that support their caches. (This effect is measured in more detail below.) In the case of Allspice, however, the limitation on its cache size is due to an artifact of the memory mapping hardware on the Sun4. The file cache uses hardware page map entries, and if it gets too large it can cause extreme contention for the few remaining map entries. Its cache is fixed arbitrarily at about 80 meg because of this. Even with this limitation, the cache on Allspice is 10 times the size of the cache on Mint and Oregano, and measurements presented in Section 8.4.3 indicate that a server needs a large cache like this for the cache to be really effective.

8.5.2. Other Storage Overhead

One of the negative lessons we learned is that the file servers can use up a considerable amount of their memory in data structures that support the file system. The data structures include stream and object descriptors described in Chapter 3, buffer space for RPC server processes, hash table buckets, and other supporting data structures. The total kernel size and the number of stream and object descriptors were measured, but a complete accounting of a file server's memory use was not attempted.

A complete breakdown of the object descriptors kept by both clients and servers is given in Table 8-10. The table gives the average number of object descriptors of various types, as well as the average cache size. The top row of the table gives the size of each

Average Object Descriptor Usage, August-September '89									
Host	Stream (100)	File (268)	RmtFile (216)	Pdev (288)	Ctrl (140)	Dev (136)	Rmt (72)	Cache Bytes	State Bytes
Mint	2722	927	130	17	250	6	1	9498624	589500
Oregano	956	1337	44	60	21	9	0	8806400	484864
Basil	181	0	255	27	11	5	9	1232896	83824
Fenugreek	217	0	520	31	12	5	5	3575808	145668
Mace	177	0	506	25	9	5	2	1421312	136280
Murder	210	72	1879	33	12	6	2	6873088	458304
Mustard	231	0	302	34	14	4	4	1679360	100916
Nutmeg	157	0	225	25	8	4	0	1298432	73164
Paprika	199	0	390	30	11	5	3	2486272	115216
Sage	213	0	625	32	12	4	5	3260416	168100
Sassafras	93	0	239	15	5	4	0	2187264	66488
Sloth	207	0	421	32	12	5	8	1015808	123788
Thyme	190	0	447	31	11	5	1	6209536	126772
AveSun3	189	7	528	29	11	5	4	2840018	145320
AveSun3*	187	0	393	28	11	5	4	2436711	114022
Cardamom	143	0	359	32	8	4	2	5992448	102868
Forgery	237	0	571	53	15	4	6	6332416	165376
Hijack	174	0	409	38	10	5	1	6017024	118840
Kvetching	308	0	641	58	14	4	3	4898816	188680
Parsley	119	0	440	26	7	4	3	6828032	116168
Pepper	69	0	406	12	5	4	1	6483968	99368
Piquante	94	0	400	19	7	4	1	5238784	102868
Piracy	72	0	176	14	4	4	0	1691648	50352
Pride	141	0	753	31	8	4	1	5431296	187412
Subversion	129	0	326	30	9	4	1	6172672	93832
Violence	122	0	284	27	7	4	2	5578752	82988
AveDS3100	146	0	433	31	9	4	2	5515078	118977
AvgClient	167	3	481	30	10	4	3	4177548	132149
AvgClient*	165	0	414	30	9	4	3	4049189	116618

Table 8-10. The average number of object descriptors during August and September 1989. The first row indicates the size of the descriptor in bytes. The last column gives the size of all the host's descriptors together. For comparison, the average cache size of the host is also given. The other columns give a raw count of the number of object descriptors of different types. ''Stream'' counts stream descriptors, including the shadow stream descriptors on the servers. ''File'' is the file descriptors kept by the file servers, and ''RmtFile'' is the remote file descriptor kept by the clients. ''Ctrl'' is a per-pseudo-device descriptor, and there is one on both the file server and the I/O server. ''Pdev'' is a per-pseudo-device-connection descriptor, and these are only on the I/O server. ''Dev'' is device descriptors. ''Rmt'' is remote descriptors, either remote devices or remote pseudo-devices.
* Murder is factored out of the starred (*) averages in order to remove the skew introduced by its nightly scans of the file system when doing tape backup.

type of descriptor. The last column gives the average state size, which is obtained by multiplying the descriptor size by the average number of descriptors and summing over all descriptor types. The average cache size is given for comparison. On average, the amount of memory used for object descriptors is not that great. The file servers average around 0.5 Mbytes of descriptors, and the clients average over 0.1 Mbytes of descriptors. A client workstation averages about 180 open I/O streams at any given time. Of these, about 70 are related to pseudo-devices: about 10 control streams to server processes, and about 30 request-response streams (‘‘Pdev’’) between clients and servers. (There are two streams for each ‘‘Pdev’’ descriptor, one for the client and one for the server.) About 90 streams are to files, and there are a handful of streams to devices and remote devices or pseudo-devices. The large number of streams to files is due in part to the I/O streams to program images (executable files) and paging files.

The kernel size and cache size for the root file server over a four month period are given in Table 8-11. The kernel size includes all code and data in the kernel, but it does not include the cache blocks. Mint has 16 Mbytes of main memory, and on average its kernel occupies about 4 Mbytes and its file cache about 9 Mbytes. For comparison, the kernel code and data of a diskless client occupies about 1.1 Mbytes. Note that the maximum observed cache size plus the maximum observed kernel size is greater than the 16 Mbytes available on Mint. The server’s memory usage increases over time as it accumulates more state information, and this reduces the memory available to the cache. Table 8-11 also gives the number of stream and file descriptors maintained by the root server, and the amount of memory they occupy. The growth of the network is also evident in the

Mint’s Kernel Size vs. Cache Size and Object Descriptors

Month	Streams		Files		Cache Meg		Kernel Meg	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
JUL	0.20	0.46	0.26	0.59	9.22	11.80	3.67	4.77
	2116	4860	878	1982				
AUG	0.20	0.49	0.26	0.58	8.99	11.57	3.78	5.20
	2066	5149	867	1962				
SEP	0.27	0.76	0.28	0.60	9.07	11.36	4.07	5.62
	2797	7981	925	2006				
OCT	0.23	0.68	0.26	0.59	8.98	11.16	3.96	5.48
	2421	7101	870	1973				

Table 8-11. The top row for each month gives the size in megabytes of various components of the kernel. ‘‘Streams’’ is stream descriptors. ‘‘Files’’ is file descriptors. ‘‘Cache’’ is the size of the file cache, and ‘‘Kernel’’ is the size of the kernel’s code and data. ‘‘Kernel’’ includes the ‘‘Streams’’ and ‘‘Files’’ space, but not the ‘‘Cache’’ space. The bottom row for each month gives the average and maximum number of object descriptors. A stream descriptor is 100 bytes (!) and a file descriptor is 312 bytes.
Note: The maximums for the different fields did not necessarily occur at the same time.

increase in the number of stream descriptors from July to October.

A comparison of the kernel sizes given in Table 8-11 to the amount of memory used for stream and file descriptors indicates that there is still a considerable amount of memory unaccounted for. There are two main contributions to the extra memory used by the file server's kernel. First, there is a ‘‘high-water’’ effect in the kernel's memory allocator. We use a binned allocator that keeps blocks of memory (‘‘bins’’) reserved for objects of a given size. The allocator adds chunks of real memory to its bins, and then it allocates from its bins. It does not attempt to coalesce memory or return chunks of memory to the rest of the system. Thus, the real memory behind each bin will remain at a high-water mark associated with the peak demand for objects of that size.

A second significant use of memory comes from the RPC server processes of the file servers. Packet handling is optimized by keeping pre-allocated buffer space for each server process, about 20 kbytes. Also, each kernel process has a private stack that is 16 kbytes. A Sprite kernel starts with 2 RPC server processes, and more are created upon demand. A file server can have up to 50 RPC server processes, which can account for 1.8 Mbytes in stacks and buffer space. This memory use could be reduced by introducing a shared buffer pool, or setting the limit below 50 server processes. This limit is somewhat arbitrary because the server processes are multiplexed among clients.

8.6. Conclusion

This chapter has reviewed the Sprite caching system and reported on its performance when supporting day-to-day work in our user community. The performance results indicate that the caches on diskless clients are effective in satisfying read requests (miss traffic is about 36%) and in eliminating write-backs to the servers (the write traffic ratio is about 53%). It appears that adding memory to a Sprite client reduces paging activity and increases the effectiveness of the file caches, as expected. However, because the client caches are so effective, the caches on the servers need to be rather large in order to reduce their disk traffic. When comparing server cache size to its disk storage, it appears that a server cache that is a few percentage of its disk capacity can be effective in cutting disk traffic.

The straight-forward approach to cache consistency, which is to disable client caches during concurrent write sharing, has worked out fine in our environment. There is very little consistency-related traffic between the servers and clients. In less than 1% of the opens did the servers have to issue cache control messages, so the approach does not add much overhead. Where there is concurrent write sharing the files are cached in the main memory of the server, and accesses to the shared data is not too expensive. Traffic to uncachable files (mainly the shared migration database) was significant. It accounted for 8% of the opens, 4% of the data read from the servers, and 11% of the data written.

A few interesting negative results stand out. First, our file servers generate a lot of traffic to their file descriptors, often just to update the access time attribute that is stored there. This traffic is an artifact our disk sub-system, which has never been tuned. Current research in Sprite focuses on log-structured file systems which should address this problem. Second, paging activity on a client with a small memory can dominate its regular I/O traffic. This traffic does not reflect a problem with our caching system, however, and it highlights the benefit of putting large memories on workstations.

Finally, the one negative result that is a function of the caching system concerns the amount of state maintained by the servers. File servers can keep a considerable amount of data structures to support the file system, enough to limit the size of their file system caches. This limit does not have a large performance impact as yet, mainly because the client caches are so effective. However, it implies that servers need a lot of memory, not just for caching data but also to maintain the data structures that support their clients' caches.

CHAPTER 9

Conclusion

9.1. Introduction

This chapter reviews the important contributions of this dissertation. The general approach of this dissertation is to use a shared network file system as the base for a distributed system. The central role of the file system requires a number of novel approachs to the problems of naming, remote access, and state management. These issues are discussed below.

9.2. The Shared File System

The Sprite distributed file system provides access to devices and services, as well as files. The file system's mechanisms for naming, remote access, and state management are applied to all of these resources. The file system acts as a name service, in contrast to other distributed systems that introduce a separate network name service. There are two primary advantages of this approach. First, users and applications are presented with the same system model that they have in a stand-alone system. Resources are accessed via the standard open-close, read-write file system interface regardless of their type and network location. The complexity of the system model is not increased by the underlying distribution of the system. The second advantage of this approach is that the operating system mechanisms provided to support remote file access are reused when accessing devices and services. Implementation of these mechanisms inside the operating system kernel allows for efficient sharing of the mechanisms. Furthermore, by reusing the file system mechanisms instead of adding a separate name service, file access is optimized because only a single server is involved.

Remote device access and remote service access create problems that are not present with remote file access. First, devices and arbitrary user-implemented services have indefinite service times, while file accesses have bounded service times. Unlike other distributed file systems, the Sprite file system has general remote waiting and synchronization facilities that account for indefinite service times. Long-term waiting is done without using up critical resources on the servers. The **select** operation can be used to wait on any collection of files, devices, and services, regardless of their location in the distributed system. Another problem created by the addition of devices and services is that these resources are not co-located with their name; the I/O server for a device or service can be different than the file server that implements their name. The internal software architecture cleanly distinguishes its naming and I/O interfaces so that different servers can implement these interfaces.

The distributed name space is implemented with a light-weight naming protocol that is unique to Sprite. Each host's view of the name space is defined by a prefix table that is maintained as a cache. Remote links in the name space trigger the addition of new prefixes to a host's prefix table, and the server for a prefix is located using a broadcast protocol. There are two primary advantages of Sprite's distributed naming mechanism. First, it reuses the directory lookup mechanisms of the file servers so there is no duplication of effort between a name service and the file service. Second, the system is decentralized; no host has to have a complete picture of the system configuration. The system adapts to changes in the configuration automatically so it is easy to manage the system as it expands. In contrast to other distributed naming systems, Sprite's naming system is optimized towards the local area network environment by eliminating globally replicated configuration information used in larger scale naming systems.

9.3. Distributed State Management

While Sprite presents a simple system model to users and applications, internally it must solve some difficult problems in distributed state management. Unlike many systems that use ''stateless'' servers for simplicity, Sprite servers keep track of how their resources are being used so they can provide high-performance services and retain the semantics of a stand-alone system. There are three interrelated features of Sprite that contribute to the complexity of its state management: data caching in the main memory of diskless clients, migration of actively executing processes among hosts in the network, and automatic recovery from the failure of hosts.

Sprite's caching system motivates the need to keep accurate state about the use of resources. The caching system provides high performance access to file data, yet accurate state about the caching system is required so the servers can maintain the consistency of cache data. While the caching system was originally described elsewhere [Nelson88a] [Nelson88b], this dissertation describes how the state that supports the caching is maintained efficiently during normal operations, during the migration of processes between hosts, and across server failures. Additionally, this dissertation presents a follow-on performance study of the caching system as it is used to support real users.

The servers protect their state by trusting their clients to keep redundant copies of the state. The servers, in turn, maintain their own state on a per-client basis so they can clean up after clients fail. The state includes a record of open I/O streams, as well as caching information. If a server fails, it can rebuild its state with the help of its clients through a state recovery protocol. An important property of the state recovery protocol is that it is *idempotent*; it can be invoked by the clients at any time and it will always attempt to reconcile the server's state with that of the clients. Idempotency is important because network partitions can lead to unexpected inconsistencies between the state on the client and the server.

Maintaining a server's state is complicated by process migration and the concurrency present in the system. I/O streams can be shared by processes which can migrate to different hosts, and distributed state changes have to be made to account for this. Processes can operate on a stream concurrently, so the distributed state changes have to be carefully synchronized. This dissertation describes a callback scheme to coordinate the distributed state changes with a deadlock-free algorithm. Migration also creates a problem with shared I/O streams because processes on different hosts can end up sharing an I/O stream and its current access position. The system uses shadow stream descriptors on the I/O server that keep stream offset when a stream is shared among processes on different hosts.

9.4. Extensibility

Sprite takes a novel approach to system extensibility. User-level services are transparently integrated into the distributed file system; they appear as device-like objects (pseudo-devices) or as whole sub-trees of the file system (pseudo-file-systems). Pseudo-devices are used to implement network protocols, terminal emulators, and the display server for the window system. Pseudo-file-systems are used to provide access to foreign kinds of file systems from within Sprite. This approach is higher-level than in a message-passing system that just provides a simple name service and a low-level communication protocol. There are additional features such as remote synchronization and state management that are provided by the distributed file system and available to the user-level services. This approach is in contrast with the commonly found approach of moving major system services like the file system out of the operating system kernel. Instead, the file system implementation is kernel-resident for efficiency and so that its mechanisms can be shared by user-level services.

9.5. Future Work

This dissertation has addressed a broad range of issues that concern a shared distributed file system for a local area network environment. There are few areas in which further work is possible. First, the maximum size of the system is still unknown. We would like to support system of up to 500 clients and their associated servers, but we have only had experience with a system one tenth this size. A full-sized system will stress the limits of our fully-shared file system. In particular, the broadcast-based naming protocol will have to be augmented to deal with a local-area *inter*network in which broadcast may not reach all hosts. Efficiently managing the system's state as the system grows may also pose difficulties. Already our file servers devote a considerable amount of memory to supporting data structures as well as their file caches.

Regrettably, the extension of the caching and recovery systems to include pseudo-devices and pseudo-file-systems has not been implemented, yet. (It's just a small matter of programming!) Using the cache for our NFS pseudo-file-system, for example, would improve access times considerably. The recovery system could also be useful to a variety of user-level services. With these extensions in place, a user-level service would be a full partner with the kernel-resident services.

9.6. Conclusion

Several key ideas are presented in this dissertation. First, the fully-shared file system takes you most of the way towards a well integrated distributed system. Cooperative work is easy, workstations are just ‘‘terminals’’ that provide access to the computing environment, and the administrative effort is comparable to that in a large timesharing system instead of many independent systems. Second, high performance and good reliability are possible using techniques that exploit large memories (for caching) and redundancy in the distributed system (for recoverability). With this approach, the best way to improve the performance of your workstation is to add more memory as opposed to adding a local disk. Larger memories reduce paging by the VM system and increase the effectiveness of the file cache. Local disks add heat, noise, and administrative hassles, and economies of scale argue for keeping large disks on shared file servers. Finally, while extensibility of the system is important, it is also important to keep system extensions well integrated into the computing environment. The pseudo-device and pseudo-file-system mechanisms achieve this goal, plus they extend features built into the kernel to user-level services, in particular the distributed name space and remote access.

APPENDIX C

Sprite RPC Protocol Summary

1. Protocol Overview

This is a brief description of the Sprite Remote Procedure Call (RPC) network protocol. The protocol is used by the Sprite Operation System kernel for communication with other Sprite kernels. It is modeled after the Birrell-Nelson protocol that uses implicit acknowlegdements so that in the normal case only two packets are exchanged for each RPC. Fragmentation is also implemented to improve performance of large data transfers, and a partial retransmission scheme is used if fragments are lost. The description given here includes a detailed look at the format of the RPC packet header and its relation to other protocols. A more detailed description can be found in [Welch86a].

The Sprite RPC protocol is used for a request-response style of communication between the operating systems on two Sprite hosts. One host, called the ''client'', issues a request to the other, called the ''server'', who responds to the request by taking some action and returning a response. The technique of implicitly acknowledging requests and responses [Birrell84] is used so that normally only two messages are exchanged for each RPC. The server's response message implicitly acknowledges the reciept of the client's request messages, and the client's next request message implicitly acknowledges the server's previous reply.

There is a ''soft binding'' between clients and servers in order to implement the implicit acknowledgment scheme. Each Sprite host has a number of RPC client *channels* (or ports), and each Sprite host has a set of RPC server processes identified by a port number. Initially clients do not specify a server port and the server assigns a free server process to a new request. The port number of the server is returned to the client and the client directs subsequent requests *from that channel* to the same server process. Thus there is a sequence of RPCs between a given client channel and a given server process. The soft binding between the client channel and the server process is broken by the server when it does not receive a new request from the client after a threashold period of time. To close the connection it sends ''close-acknowledge'' message to the client to make sure that the last reply was successfully received.

Request and reply messages are allowed to be larger than the maximum transmission size of the underlying transport protocol (either IP or ethernet). Large messages are divided into fragments, and the same implicit acknowledgement scheme is used. Upon reciept of all the fragments of a request, for example, the server takes an action and returns a single reply message. The individual fragments are not individually acknowledged.

The RPC protocol uses an unreliable datagram protocol to deliver messages, either raw ethernet packets or the IP datagram protocol. In the case of packet loss the client, or requesting host, takes the active role to ensure that the RPC completes. It sets up a timeout and will retransmit its request if it does not get a response before the timeout period expires. If its request is fragmented, only the last fragment is retransmitted. The server responds to retransmissions with explicit acknowledgment messages. If the request is fragmented the server's acknowledgment message indicates which fragments have been received. If a reply message is fragmented and some fragments are lost, the client retransmits its request and the packet header indicates which fragments of the reply have been received. After these ''partial acknowledments'' the sender of a fragmented message will retransmit only the lost fragments.

Note that the protocol does not rely on fragmentation by the underlying transport protocol. There are two reasons for this. The first is that by providing its own fragmentation the RPC protocol can provide efficient transfer of large blocks of data without using a specific transport protocol. Second, datagram protocols cannot implement partial retransmission in the case of lost fragments. They discard the whole packet if any fragments are lost. This is undesirable when communicating between hosts of different speeds where overrun at the slow host is not uncommon. The RPC protocol header includes a desired inter-fragment delay that hosts use to throttle down their transmit rate when sending fragmented messages to slower hosts.

The protocol header has some support for crash detection and graceful handling of server reboots. The header includes a bootID that is reset to the network time each time a host boots up. This allows other hosts to detect reboots by noting a change in the bootID. There is also a flag bit in the header that hosts set while they are booting. This indicates to other hosts that the booting hosts is active but not yet ready to service requests. This is useful for servers that take a number of minutes to check the consistency of their disks before providing service.

The data transferred in the request and response messages is divided into two parts. The first part is called the ''parameter'' area, and it is used to transmit integers and other simple data structures. The second part is called the ''data'' area and is usually a larger block uninterpreted data. This format is oriented towards procedures that have a few small parameters and one large block of data. This format also allows us to implement simplified byte swapping to handle communication between hosts with different architectures. A field in the RPC packet header identifies the sender's byte ordering, and the receiver is responsible for byte swapping the RPC packet header and the parameter area. These two parts of the packet are restricted to contain only integers so they can be byte swapped without knowing their exact contents.

2. RPC Packet Format

A packet sent by the RPC protocol is divided into four parts, the datagram header, the RPC header, the RPC parameter area, and the RPC data area. This whole packet may be futher encapsulated by the link level transport, i.e. ethernet. Table 8-2 defines the

RPC Packet Format		
Bytes	**Field**	**Description**
0-3	version	Version Number/Byte Ordering
4-7	flags	Type and flag bits
8-11	clientID	Client Sprite ID
12-15	serverID	Server Sprite ID
16-19	channel	Client channel (port) number
20-23	serverHint	Server index (port)
24-27	bootID	Sender's boot timestamp
28-31	ID	RPC sequence number
32-35	delay	Inter-fragment delay
36-39	numFrags	Number of fragments in this message
40-43	fragMask	Bitmask identifying fragment
44-47	command	RPC procedure identifier
48-51	paramSize	Number of bytes in parameter area
52-55	paramOffset	Offset of this fragment's parameters
56-59	dataSize	Number of bytes in data area
60-63	dataOffset	Offset of this fragment's data

Table C-1. There are 16 4-byte integer fields in an RPC packet header. This is followed by two variable length regions called the ''parameter'' and ''data'' areas. The whole RPC packet is further encapsulated in a lower-level transport format, either raw Ethernet or IP.

RPC Type and Flag Bits		
Bit	**Value**	**Description**
RPC_TYPE	0xff00	Mask for type bits
RPC_REQUEST	0x0100	Request message
RPC_ACK	0x0200	Acknowledgment message
RPC_REPLY	0x0400	Reply message
RPC_ECHO	0x0800	Special low-level echo message
RPC_FLAG	0x00ff	Mask for flag bits
RPC_PLSACK	0x0001	Please acknowledge REQUEST
RPC_LASTFRAG	0x0002	Last fragment in REQUEST/REPLY, or partial ACK.
RPC_CLOSE	0x0004	Close connection between client and server (ACK)
RPC_ERROR	0x0008	REPLY contains error code in command field
RPC_SERVER	0x0010	Message to server (REQUEST or ACK)
RPC_LAST_REC	0x0020	(Internal) Marks end of circular trace
RPC_NOT_ACTIVE	0x0040	Sender is not fully active (REQUEST/ACK/REPLY)

Table C-2. The flags field in the RPC packet header contains these flag and type bits. The types are mutually exclusive, while the flags can be found in the packets of the indicated type.

format of the RPC protocol header. Note that all fields are 4 byte integers to facilitate byte swapping between hosts of different architectures. All packets (request, replies, and acknowledments) sent by the RPC protocol have this same header. The parameter and/or data area may be of zero length.

The flags field contains both packet type information (request, reply, ack) and some flags (error reply, please acknowledge, close connection, to server, host not active). The values for the flags are defined in Table C-2. The explaination of each flag indicates the type of packet in which it occurs.

The remaining packet fields are described below.

version — This field defines the RPC version number and it is also used to detect byte ordering differences between the client and the server. The currently defined version numbers are:

RPC_NATIVE_VERSION	0x0f0e0003
RPC_SWAPPED_VERSION	0x03000e0f

If a receiver gets a packet with the byte-swapped version number it has to byte-swap the RPC header and the parameter block. This can be done by treating each field in the header as an integer, and by treating the parameter block as an array of integers. The data block remains uninterpreted by the low levels of the RPC system.

flags — The flag bits are defined above in Table C-2.

clientID — Sprite host ID of client host. Sprite host IDs are distinct from the low-level host IDs used by the datagram protocol (i.e. ethernet or IP).

serverID — Sprite host ID of server host.

channel — The client channel number. Each Sprite hosts has a small number of channels (8) which provide for concurrent communication.

serverHint — A hint as to the server process port.

bootID — Boot-time timestamp. Set to network time upon booting.

ID — RPC sequence number. The same on all packets concerning a single RPC.

delay — Desired interfragment delay, in microseconds. This is used when sending fragmented messages to slower hosts to introduce a time delay between transmission of packet fragments.

numFrags — Number of fragments in the message. 0 (zero) indicates no fragmenting. The maximum number of fragments supported in the implementation is currently 16, but this field allows up to 32 fragments.

fragMask — Bitmask indicating which fragment of a REQUEST or REPLY this is. If no fragmenting then this field has to be 0 (zero). In ACK packets this mask indicates what fragments have been received.

command — Command (procedure) identifier for REQUEST. For REPLY messages with the ERROR bit set this field is overloaded with an error code.

paramSize — The number of bytes in the first data area.

paramOffset	The offset of this piece of the parameters within the whole parameter block.
dataSize	The number of bytes in the second data area.
dataOffset	The offset of this piece of the data within the whole data block.

3. Protocol Layering

The RPC protocol is designed to use any datagram protocol for transport between hosts. The format of the datagram header is not important to the RPC protocol itself, although the routing module in the Sprite kernel will examine the datagram header to learn the correspondence between Sprite host IDs and the addresses used by the transport protocol. When used with the Internet Datagram Protocol (IP), the RPC header, parameters, and data would follow the IP header immediately. We have been assigned Internet protocol number 90 for implementations layered on top of the IP protocol.

Bytes 0-31	IP header (with no options)
Bytes 32-95	RPC header
Bytes 96+	RPC parameters and data, if any

4. Source Code References

The RPC protocol implementation can be found in the Sprite kernel sources. The directory is ‘‘/sprite/src/kernel/rpc’’. The definitions of the packet formats given here come from the file ‘‘rpcPacket.h’’. Statistics that are collected by the on-line RPC system are defined in ‘‘rpcCltStat.h’’ and ‘‘rpcSrvStat.h’’. These statistics are available via the **rawstat** and **rpcstat** Sprite commands.

APPENDIX D

The Sprite RPC Interface

1. Introduction

This is a brief summary of the network interface to Sprite. This is done via a description of each RPC used in the implementation. Most of the calls have to do with the file system, although there are a few related to process migration, signals, and remote waiting. The parameters of each RPC are specified, and a few words about the use of the RPC are given. This is only a terse reference guide; there is not necessarily justification or explaination included in the descriptions.

It is important to understand that this RPC interface is between two operating system kernels. The terms ‘‘client’’ and ‘‘server’’ refer to instances of the kernel acting in these roles. The term ‘‘application process’’ will be used when necessary to refer to the process triggering the use of an RPC.

1.1. The RPC Protocol

It is useful to review the appendix on the RPC protocol itself. The main thing to understand is that the information included in each RPC request and reply is broken into three parts, a standard RPC header, a parameter area, and a data area. The actual size of the parameter and data areas are extracted from the packet header and reported to the stub procedures. The ID of the client is also reported to the server-side stub procedures. The stubs have to deal with the distinct parameter and data areas. The parameter area is restricted to contain only integers, while the data area can contain arbitrary data. The low-levels of the RPC protocol handle communication between hosts of different byte order by automatically byteswapping the RPC header and the parameter area when a packet is recieved from a host with a different byte order. This means that both the client and server stubs can treat the parameter area as a C structure (of integers), and not worry about alignment problems. If the data area is used for other than file data or strings, then the stub, or even a higher-level procedure, has to do its own byteswapping.

Lastly, there is a return code from each RPC, with SUCCESS (zero) meaning successful completion. Sprite return codes are defined in the file ‘‘/sprite/lib/include/status.h’’.

The complete set of RPCs used in the Sprite implementation is given in Table D-1. The number associated with the RPC is the procedure number used in the RPC packet header. The RPCs are described in more detail below.

Sprite Remote Procedure Calls		
Procedure	#	Description
ECHO_1	1	Echo. Performed by server's interrupt handler (unused).
ECHO_2	2	Echo. Performed by Rpc_Server process.
SEND	3	Send. Like Echo, but data only transferred to server.
RECEIVE	4	Receive. Data only transferred back to client.
GETTIME	5	Broadcast RPC to get the current time.
FS_PREFIX	6	Broadcast RPC to find prefix server.
FS_OPEN	7	Open a file system object by name.
FS_READ	8	Read data from a file system object.
FS_WRITE	9	Write data to a file system object.
FS_CLOSE	10	Close an I/O stream to a file system object.
FS_UNLINK	11	Remove the name of an object.
FS_RENAME	12	Change the name of an object.
FS_MKDIR	13	Create a directory.
FS_RMDIR	14	Remove a directory.
FS_MKDEV	15	Make a special device file.
FS_LINK	16	Make a directory reference to an existing object.
FS_SYM_LINK	17	Make a symbolic link to an existing object.
FS_GET_ATTR	18	Get the attributes of the object behind an I/O stream.
FS_SET_ATTR	19	Set the attributes of the object behind an I/O stream.
FS_GET_ATTR_PATH	20	Get the attributes of a named object.
FS_SET_ATTR_PATH	21	Set the attributes of a named object.
FS_GET_IO_ATTR	22	Get the attributes kept by the I/O server.
FS_SET_IO_ATTR	23	Set the attributes kept by the I/O server.
FS_DEV_OPEN	24	Complete the open of a remote device or pseudo-device.
FS_SELECT	25	Query the status of a device or pseudo-device.
FS_IO_CONTROL	26	Perform an object-specific operation.
FS_CONSIST	27	Request that cache consistency action be performed.
FS_CONSIST_REPLY	28	Acknowledgement that consistency action completed.
FS_COPY_BLOCK	29	Copy a block of a swap file.
FS_MIGRATE	30	Tell I/O server that an I/O stream has migrated.
FS_RELEASE	31	Tell source of migration to release I/O stream.
FS_REOPEN	32	Recover the state about an I/O stream.
FS_RECOVERY	33	Signal that recovery actions have completed.
FS_DOMAIN_INFO	34	Return information about a file system domain.
PROC_MIG_COMMAND	35	Used to transfer process state during migration.
PROC_REMOTE_CALL	36	Used to forward system call to the home node.
PROC_REMOTE_WAIT	37	Used to synchonize exit of migrated process.
PROC_GETPCB	38	Return process table entry for migrated process.
REMOTE_WAKEUP	39	Wakeup a remote process.
SIG_SEND	40	Issue a signal to a remote process.

Table D-1. The Remote Procedure Calls that are used in the Sprite implementation.

1.2. Source Code References

The RPC parameters described below are given as C typedefs. These have been extracted from the C source code of the Sprite implementation. It may help to consult the code in order to fully understand the use of a particular RPC. Most of the RPC stubs are organized so the client and server stub for a given RPC are together along with the definition of their parameters. However, these definitions also include structures that are common to the rest of the implementation and may be defined elsewhere. Table D-2 has a list of the ‘‘.c’’ files that contain most of the RPC stubs and the ‘‘.h’’ files that contain most of the relavent typedefs. Finally, note that these structures are passed around within the kernel as well as between kernels using RPC, and there are some structure fields that are not valid during an RPC.

2. ECHO_2 This echoes data off another host. This operation is handled by an Rpc_Server process, so it exercises the full execution path involved in a regular RPC. (The unsupported ECHO_1 was used to echo off the interrupt handler.) This is used for benchmarking the RPC system, and it is also used by the recovery module to verify that another host is up. Equal amounts of data are transferred in both directions. The data is uninterpreted, and it is put in the data area of the RPC packet.

Important Source Files	
fs/fs.h	Basic FS definitions.
fs/fsNameOps.h	Definitions for the naming interface.
fsrmt/fsNameOpsInt.h	More definitions for the naming interface.
fsrmt/fsrmtInt.h	Definitions for the I/O interface.
fsrmt/fsSpriteDomain.c	RPC stubs for the naming interface.
fsrmt/fsSpriteIO.c	RPC stubs for the I/O interface
fsrmt/fsRmtAttributes.c	RPC stubs for attribute handling.
fsrmt/fsRmtDevice.	RPC stubs for remote devices.
fsrmt/fsRmtMigrate.c	RPC stubs for migraiton.
fsio/fsStream.c	RPC stub for migration callback.
fsconsist/fsCacheConsist.c	RPC stubs for cache consistency.
fsutil/fsRecovery.c	RPC stubs for state recovery.

Table D-2. Important source files concerning the RPC interface to Sprite. The file names are relative to the ‘‘/sprite/src/kernel’’ directory. This list isn’t guaranteed to be complete. There are likely to be other definition files needed to fully resolve the parameter definitions of an RPC. (All required definitions are given in this appendix.)

3. SEND This transmints data to another host. This is just used for benchmarking the RPC system. Data is only transferred one way, from the client to the server.

4. RECEIVE This receives data from another host. This is only used for benchmarking the RPC system.

5. GETTIME This is a broadcast RPC used to get the time-of-day. This is used by hosts during boot strap in order to set their clocks. The clock value is also used to set their rpcBootID, which is included in the header of all subsequent RPC packets. This is the first RPC done by a Sprite host, and the bootID in its RPC header is zero. The parameters and data areas of the RPC request are empty. The parameter area of the reply contains the following structure:

```
typedef struct RpcTimeReturn {
        Time    time;
        int     timeZoneMinutes;
        int     timeZoneDST;
} RpcTimeReturn;
```

The Time data type is defined as:

```
typedef struct Time {
        int     seconds;
        int     microseconds;
} Time;
```

The time is the number of seconds since Jan 1, 1970 in universal (GMT) time. The timeZoneMinutes is the offset of the local timezone from GMT. The timeZoneDST is a flag indicating if daylight savings time is allowed in the timezone (not if it is in effect at the current date).

6. FS_PREFIX This is a broadcast RPC used to locate the server for a prefix. The request parameter area is empty. The request data area contains the null-terminated prefix, whose length can be determined from the RPC packet header's dataLength field. The reply data area is empty. The reply parameter area contains the following structure:

```
typedef struct FsPrefixReplyParam {
        FsrmtUnionData      openData;
        Fs_FileID       fileID;
} FsPrefixReplyParam;
```

The fileID identfies the root directory of the *domain* identified by the prefix. The client should cache the mapping from the prefix to the fileID. It specifies this fileID as the *prefixID* in lookup RPCs (FS_OPEN, etc.). The openData is used by Sprite clients to set up internal data structures associated with an open I/O stream. The FsrmtUnionData typedef is described under FS_OPEN. The Fs_FileID is defined as follows:

```
typedef struct Fs_FileID {
        int     type;
        int     serverID;
        int     major;
```

```
    int   minor;
} Fs_FileID;
```

The serverID is the Sprite hostID of the I/O server for the object. The major and minor fields identify the object to the I/O server. The values for the type field are defined in Table D-3. With the FS_PREFIX RPC the returned type is either FSIO_RMT_FILE_STREAM for regular Sprite file systems, or FSIO_PFS_NAMING_STREAM for pseudo-file-systems.

7. FS_OPEN This is used to open a file system object in preparation for further I/O operations. The parameters to an open RPC include a pathname, which may have several components, and a file ID that indicates where the pathname starts. It also includes information about how the object will be used, and identification of the user and the client host that is doing the open. The reply to an open is one of two things. Ordinarily data is returned that is used to set up data structures for an I/O stream to the object. Depending on the type of object opened, which is specified in the reply, the reply contains different

Sprite Internal Stream Types		
#	Name	Description
0	FSIO_STREAM	For top-level stream descriptor.
1	FSIO_LCL_FILE_STREAM	Local file.
2	FSIO_RMT_FILE_STREAM	Remote file.
3	FSIO_LCL_DEVICE_STREAM	Local device.
4	FSIO_RMT_DEVICE_STREAM	Remote device.
5	FSIO_LCL_PIPE_STREAM	Anonymous pipe.
6	FSIO_RMT_PIPE_STREAM	Remote anonymous pipe.
7	FSIO_CONTROL_STREAM	Pseudo-device control stream.
8	FSIO_SERVER_STREAM	Pseudo-device server stream.
9	FSIO_LCL_PSEUDO_STREAM	Attached to a server stream.
10	FSIO_RMT_PSEUDO_STREAM	Indirectly attached to a server stream.
11	FSIO_PFS_CONTROL_STREAM	Records pseudo-file-system server.
12	FSIO_PFS_NAMING_STREAM	Pseudo-file-system naming stream.
13	FSIO_LCL_PFS_STREAM	Pseudo-file-system I/O stream.
14	FSIO_RMT_PFS_STREAM	Remote Pseudo-file-system I/O stream.
15	FSIO_RMT_CONTROL_STREAM	Fake type for GET_ATTR of a pseudo-device.
16	FSIO_PASSING_STREAM	Fake type when passing open streams.

Table D-3. The internal stream types used in the Sprite implementation. Most types have corresponding local (LCL) and remote (RMT) types. The CONTROL and SERVER streams, however, are always local because they are between the kernel and the pseudo-device or pseudo-file-system server process. The PFS_NAMING stream is always remote. It is returned in response to prefix broadcasts in the case of a pseudo-file-system. Some additional types are defined for use with the kernel's internal I/O interface, and these are included here for completeness.

data used for this purpose. However, it is also possible that the pathname leaves the server's domain, in which case a new pathname is returned to the client, and perhaps also a new prefix. These details are described in more detail below.

7.1. FS_OPEN Request Format The request data area contains a null terminated path name. The request parameter area contains the following structure:

```
typedef struct Fs_OpenArgs {
        Fs_FileID       prefixID;
        Fs_FileID       rootID;
        int     useFlags;
        int     permissions;
        int     type;
        int     clientID;
        int     migClientID;
        Fs_UserIDs      id;
} Fs_OpenArgs;
```

The prefixID specifies where the pathname begins. There are two cases for this. If the client initially has an absolute pathname, then it can match this against its prefix cache and use the fileID associated with the longest matching prefix. The prefix should be stripped off before sending the pathname to the server. If the client initially has a relative pathname, then the prefixID is the fileID associated with the current working directory. This is obtained by a previous FS_OPEN RPC on the current directory. The rootID is a prefix ID, and it is used to trap out pathnames that ascend out the root directory of a domain. It is either the same as the prefixID, or it is the ID of the prefix that identifies the domain of the current directory. Note that this supports implementation of chroot(), and it also allows servers to export prefixes that don't correspond to the ''natural'' root directory of a domain.

The permissions field contains the permission bits to set on newly created files. These are defined in Table D-4.

The type field contrains the type of object that can be opened. If the type is FS_FILE, then any type can be opened. This is the way the FS_OPEN RPC is used by the open() system call. However, FS_OPEN is also used in the implementation of readlink(), symlink(), and mknod(). For these calls specific types are indicated so that the type of the named file must match. (Warning, note the bug report concerning FS_REMOTE_LINK and FS_SYMBOLIC_LINK in the last section of this appendix.) The types are defined in Table D-5, and they correspond to types in the file descriptors kept on disk.

The clientID is the hostID of the process doing the open. The migClientID is the home node of a migrated process, which may be different than the clientID if a process has migrated. This is used when opening devices so that a migrated process can open devices on its home node. This only applies to device files with the FS_LOCALHOST_ID serverID attribute. Device files with a specific host ID (>0) for their serverID attribute always specify the device on that host. (See FS_MAKE_DEV below.)

Permission Bits	
FS_OWNER_READ	00400
FS_OWNER_WRITE	00200
FS_OWNER_EXEC	00100
FS_GROUP_READ	00040
FS_GROUP_WRITE	00020
FS_GROUP_EXEC	00010
FS_WORLD_READ	00004
FS_WORLD_WRITE	00002
FS_WORLD_EXEC	00001
FS_SET_UID	04000
FS_SET_GID	02000

Table D-4. Permission bits for the Fs_OpenArgs structure. These are octal values

File Descriptor Types	
FS_FILE	0
FS_DIRECTORY	1
FS_SYMBOLIC_LINK	2
FS_REMOTE_LINK	3
FS_DEVICE	4
(not used)	5
FS_LOCAL_PIPE	6
FS_NAMED_PIPE	7
FS_PSEUDO_DEV	8
(not used)	9
(reserved for testing)	10

Table D-5. Types for disk-resident file descriptors. Also used for the type field in the Fs_OpenArgs structure.

The id field contains the user and group IDs of the process doing the open. The Fs_UserIDs typedef is defined as follows:

```
#define FS_NUM_GROUPS      8

typedef struct Fs_UserIDs {
        int user;
        int numGroupIDs;
        int group[FS_NUM_GROUPS];
} Fs_UserIDs;
```

The useFlags indicate how the object is going to be used. Valid useFlag bits are defined in Table D-6. They are divided into two sets, those passed into the Fs_Open

Usage Flags		
Value	Name	Description
0xfff	FS_USER_FLAGS	A mask used to prevent user programs from setting kernel bits.
0x001	FS_READ	Open the object for reading.
0x002	FS_WRITE	Open the object for writing.
0x004	FS_EXECUTE	Open the object for execution, if it is a regular file, or for changing the current directory if it is a directory.
0x008	FS_APPEND	Open the object for append mode writing.
0x020	FS_PDEV_MASTER	Open a pseudo-device as the server process.
0x080	FS_PFS_MASTER	Open a remote link as the server for a pseudo-file-system.
0x200	FS_CREATE	Create a directory entry for the object if it isn't there already.
0x400	FS_TRUNC	Truncate the object after opening.
0x800	FS_EXCLUSIVE	If specified with FS_CREATE, then the open will fail if the object already exists.
0xfffff000	FS_KERNEL_FLAGS	Bits are set in this field by the client kernel.
0x00001000	FS_FOLLOW	Follow symbolic links when traversing the pathname.
0x00004000	FS_SWAP	The file is being used as a VM backing file.
0x00010000	FS_OWNERSHIP	Set when changing ownership or permission bits. The server should verify that the opening process owns the file.
0x00020000	FS_DELETE	Set when deleting a file. (FS_UNLINK RPC, not FS_OPEN.)
0x00040000	FS_LINK	Set when creating a hard link. (FS_LINK RPC, not FS_OPEN.)
0x00080000	FS_RENAME	Set during rename. (FS_RENAME RPC, not FS_OPEN.)

Table D-6. Values for the useflags field of Fs_OpenArgs and Fs_LookupArgs.

system call from user programs, and those set by the Sprite kernel for its own use. (For historical reasons the UNIX open() call is mapped to Fs_Open, and some of the flag bits differ from the UNIX O_* flags, mainly the READ and WRITE bits.)

7.2. FS_OPEN Reply Format The data area of an FS_OPEN reply may contain a returned pathname, in the case where the input pathname leaves the server's domain. This is indicated by the FS_LOOKUP_REDIRECT return code. If the returned name begins with a '/' character then it has been expanded by the server and can be matched against the client's prefix table. If the prefixLength parameter (defined below) is non-zero then the initial part of the expanded pathname is a domain prefix that should be added to the client's prefix table. The client should use the FS_PREFIX RPC to locate the domain's server. If the returned pathname begins with ''../'', then the client has to combine the returned pathname with the prefix it used for the server's domain as follows. (This means a client implementation has to remember the domain prefix associated with a current working directory.) If the prefix is ''/a/b'' and the returned pathname is ''../x/y'', then these are combined into ''/a/b/../x/y'', which further reduces to ''/a/x/y''. Note this no longer matches on the ''/a/b'' prefix.

The parameter area of the FS_OPEN reply contains data used to initialize the I/O stream data structures. The format of the rely parameters is defined as follows:

```
typedef struct FsrmtOpenResultsParam {
        int     prefixLength;
        Fs_OpenResults      openResults;
        FsrmtUnionData      openData;
} FsrmtOpenResultsParam;

typedef struct Fs_OpenResults {
        Fs_FileID      ioFileID;
        Fs_FileID      streamID;
        Fs_FileID      nameID;
        int     dataSize;
        ClientData      streamData;
} Fs_OpenResults;

typedef union FsrmtUnionData {
        Fsio_FileState fileState;
        Fsio_DeviceState      devState;
        Fspdev_State   pdevState;
} FsrmtUnionData;
```

The Fs_OpenResults contain three object identifiers, one for the object that was opened, one for the I/O stream to that object (the streamID is type FSIO_STREAM), and one for the file that names the object (the nameID is type FSIO_RMT_FILE_STREAM). For devices and pseudo-devices the file that represents the name is different than the object itself, and the nameID is used when getting the attributes of an object. The I/O stream also has an identfier because I/O streams are passed between machines during process migration. The rest of the data is type-specific and is described below.

```
typedef struct Fsio_FileState {
        int     cacheable;
        int     version;
```

```
    int    openTimeStamp;
        Fscache_Attributes attr;
        int       newUseFlags;
} Fsio_FileState;

typedef struct Fscache_Attributes {
        int       firstByte;
        int       lastByte;
        int       accessTime
        int       modifyTime;
        int       createTime;
        int       userType;
        int       permissions;
        int       uid;
        int       gid;
} Fscache_Attributes;
```

The Fsio_FileState is returned when a regular file or directory is opened. In this case the type in the ioFileID is FSIO_RMT_FILE_STREAM. The cacheable flag indicates if the client can cache the file. The version number is used to detect stale data in the client's cache. The openTimeStamp should be kept and used later when handling FS_CONSIST RPCs. (The openTimeStamp is redundant with respect to the version number, and it may be eliminated in the future.) The Fscache_Attributes are a sub-set of the file attributes stored at the file server. The lastByte, modifyTime and accessTime are updated by the client if it caches the file. These attributes are pushed back to the server at close time. (firstByte is unused. It is a vestage of a named pipe implementation.) The permission bits and ownership IDs are used with setuid and setgid programs. The newUseFlags are a modified version of the useFlags passed in the FS_OPEN request. The client stores these in its I/O stream descriptor and does consistnecy checking on subsequent I/O operations by the user-level application.

```
typedef struct Fsio_DeviceState {
        int       accessTime;
        int       modifyTime;
        Fs_FileID       streamID;
} Fsio_DeviceState;
```

The Fsio_DeviceState is returned from the file server when a device is opened. The type in the ioFileID is either FSIO_LCL_DEVICE_STREAM or FSIO_RMT_DEVICE_STREAM. In the latter case, the client will pass the Fsio_DeviceState to the I/O server in a FS_DEV_OPEN RPC. The accessTime and modifyTime are maintained at the I/O server. (Currently they are never pushed back to the file server. This is bug.) The streamID is the same as in the Fs_OpenResults, but it is included in Fsio_DeviceState so it can be passed to the I/O server.

```
typedef struct Fspdev_State {
        Fs_FileID       ctrlFileID;
```

```
    Proc_PID    procID;
        int     uid;
        Fs_FileID       streamID;
} Fspdev_State;
```

The Fspdev_State is returned when a pseudo-device is opened. The type in the ioFileID is FSIO_CONTROL_STREAM when the server process opens the pseudo-device. In this case the Fspdev_State structure is not returned.

When any other process opens the pseudo-device then the type in the ioFileID is either FSIO_LCL_PSEUDO_STREAM or FSIO_RMT_PSEUDO_STREAM. In the latter case the client passes the Fspdev_State to the I/O server with a FS_DEV_OPEN RPC. The ctrlFileID identfies the control stream of the server process. The procID and uid should be filled in by the client before the FS_DEV_OPEN RPC. The streamID is the same as that in the Fs_OpenResults, but it is included in Fspdev_State so it can be passed to the I/O server. The I/O server sets up a shadow stream descriptor that matches the client's.

When a process opens a file in a pseudo-file-system then the type in the ioFileID is FSIO_RMT_PFS_STREAM. (If the pseudo-file-system server is on the same host as the process doing the open, the the FS_OPEN RPC is not used, so the FSIO_LCL_PFS_STREAM is not seen here.)

8. FS_READ This call is used to read data from a file system object. The data area of the request message is empty. The parameter area of the request contains the following structure.

```
typedef struct FsrmtIOParam {
        Fs_FileID       fileID;
        Fs_FileID       streamID;
        Sync_RemoteWaiter waiter;
        Fs_IOParam      io;
} FsrmtIOParam;

typedef struct {
        List_Links      links;
        int     hostID;
        Proc_PID        pid;
        int     waitToken;
} Sync_RemoteWaiter;

typedef struct Fs_IOParam {
        Address         buffer;
        int     length;
        int     offset;
        int     flags;
        Proc_PID        procID;
        Proc_PID        familyID;
```

```
    int   uid;
        int      reserved;
  } Fs_IOParam;
```

The fileID and the streamID together specify the I/O stream and the object it references. These have been returned from a previous FS_OPEN RPC. The Sync_RemoteWaiter stucture contains information about the process in case the read operation would block, in which case it is used in a subsequent REMOTE_WAKEUP RPC. (The links field of this is not valid during the RPC.) The length in the Fs_IOParam indicates how much data is to be read, and the offset indicates the byte offset at which to start the read. (The buffer field is not valid during the RPC.) The flags are the flags from the I/O stream descriptor, which are derived from the useFlags in FS_OPEN. The procID, familyID, and uid are used for ownership checking by certain devices. The reserved field is currently unused, but may eventually be used for a file's version number.

The reply message for a read contains the data in the data area, and a Fs_IOReply in the parameter area.

```
typedef struct Fs_IOReply {
        int      length;
        int      flags;
        int      signal;
        int      code;
} Fs_IOReply;
```

The length indicates the number of bytes returned. The flags indicate the select state of the object. They are an or'd combination of the FS_READ, FS_WRITE, and FS_EXECUTE bits defined above. The signal and code are used to return a signal from certain kinds of devices. If non-zero, the signal field will result in that signal being sent to the application process, along with the code that modifies the signal.

If the return code of the FS_READ RPC is FS_WOULD_BLOCK, then length may be greater than or equal to zero. This indicates that less than the requested amount of data was returned, and that the I/O server has saved the Sync_RemoteWaiter information. A REMOTE_WAKEUP RPC will be generated by the I/O server when the object becomes readable.

9. FS_WRITE This is similar to the FS_READ RPC, except that the request data area contains the data to be transfered. The request parameters are the same as for FS_READ. The reply parameters are also the same. If the I/O server doesn't accept all the data transferred to it then the client will block the application process if FS_WOULD_BLOCK is returned. Otherwise a short write and a SUCCESS error code will prompt an immediate retry of the rest of the data.

There are some additional flags in the Fs_IOParam structure that pertain to writes. These are described below.

0x00100000 FS_CLIENT_CACHE_WRITE

When a client writes back data from its cache this flag is set. In this case the file server does not reset the modify time because the client maintains the modify time while caching the file.

0x02000000 FS_LAST_DIRTY_BLOCK
: This is set when a client is writing back its last block for a particular file. At the receipt of the last block a server in write-back-ASAP mode will schedule the file to be written to disk, although the RPC will return before the file actually gets to disk.

0x10000000 FS_WB_ON_LDB
: This tells the server to write through the file if it is the last dirty block. This will appear with FS_LAST_DIRTY_BLOCK. At the receipt of a block marked with this flag the server will block until the file is written through, and the RPC will return after the file is on disk.

10. FS_CLOSE When the last process using an I/O stream closes its reference, then an FS_CLOSE RPC is made to the I/O server. The request data area is empty. The request parameter area is described below. The reply message has no data or parameters, only a return code.

```
typedef struct FsRemoteCloseParams {
        Fs_FileID       fileID;
        Fs_FileID       streamID;
        Proc_PID        procID;
        int     flags;
        FsCloseData     closeData;
        int     closeDataSize;
} FsRemoteCloseParams;

typedef union FsCloseData {
        Fscache_Attributes      attrs;
} FsCloseData;
```

The fileID is the ioFileID from the FS_OPEN RPC. The streamID is also from the FS_OPEN RPC. The procID identfies the process doing the close. The flags are the flags from the stream descriptor, and they may also include the FS_LAST_DIRTY_BLOCK and FS_WB_ON_LDB flags if a file is in write-back-on-close mode. The closeData is a type-specific union used to propagate attributes back to the file server at close time. Currently this is only implemented for regular files, in which case the closeData the Fscache_Attributes already described. (It should also be implemented for devices, but the file server is not contacted at close time for devices.) The closeDataSize is the number of valid bytes in the closeData union.

11. FS_UNLINK This RPC is used to remove a directory entry for an object. When the last directory entry that references an object is removed, the underlying object is deleted. However, if the object is still referneced by an open I/O stream then the deletion is postponed until the I/O stream is closed. The request data area contains a null terminated pathname. The request parameter area contains an Fs_LookupArgs record, which is a sub-set of the Fs_OpenArgs record previously defined.

```
typedef struct Fs_LookupArgs {
        Fs_FileID       prefixID;
        Fs_FileID       rootID;
        int     useFlags;
        Fs_UserIDs      id;
        int     clientID;
        int     migClientID;
} Fs_LookupArgs;
```

The reply to an RPC_UNLINK is ordinarily only a return code. However, as with all operations on pathnames, the server may return a new pathname if the input pathname leaves its domain. In this case the return code is FS_LOOKUP_REDIRECT, and the return data area contains the new pathname (see NB below), and the return parameter area contains an integer indicating the length of the prefix embedded in the returned pathname, or zero.
NB: For implementation reasons, the return data area also contains room for the prefix length (4 bytes) in front of the returned pathname. See FS_OPEN for a fuller explaination of FS_LOOKUP_REDIRECT.

12. FS_RENAME This is an operation on two pathnames, and it is used to change the name of an object in the file system. The request parameters include the Fs_LookupArgs previously defined, plus another fileID for the prefix of the second pathname. The data area contains two pathnames, and currently this is defined as two maximum length character arrays.

```
typedef struct Fs_2PathParams {
        Fs_LookupArgs   lookup;
        Fs_FileID       prefixID2;
} Fs_2PathParams;

#define FS_MAX_PATH_NAME_LENGTH         1024

typedef struct Fs_2PathData {
        char    path1[FS_MAX_PATH_NAME_LENGTH];
        char    path2[FS_MAX_PATH_NAME_LENGTH];
} Fs_2PathData;
```

There can also be a pathname redirection during a FS_RENAME, in which case the return code is FS_LOOKUP_REDIRECT, and the return data area has the returned pathname. Again, for implementation reasons two fields in the parameter area (Fs_2PathReply) are repeated in the data area before the returned pathname (Fs_2PathRedirectInfo). The name1ErrorP flag that is returned indicates whether the error code applies to the first name (if name1ErrorP is non-zero), or to the second pathname. (Please excuse the fact that prefixLength and name1ErrorP occur in opposite orders! ugh. The Fs_2PathRedirectInfo is passed around the kernel internally, and these two fields are copied into the parameter area so they get byteswapped correctly.)

```
typedef struct Fs_2PathReply {
```

```
   int   prefixLength;
         Boolean        name1ErrorP;
} Fs_2PathReply;

typedef struct Fs_2PathRedirectInfo {
         int name1ErrorP;
         int      prefixLength;
         char fileName[FS_MAX_PATH_NAME_LENGTH];
} Fs_2PathRedirectInfo;
```

Note that redirection makes lookups involving two pathnames slightly more complicated than an operation on a single pathname. A Sprite client will use its prefix cache to get the prefixFileID for both pathnames, and they may specify different servers. The client always directs the FS_RENAME (or FS_LINK) to the server for the first pathname. If the second pathname is also in the same domain then the FS_RENAME (or FS_LINK) can complete with a single RPC. If the first pathname leaves the server's domain, the server returns FS_WOULD_BLOCK and sets name1ErrorP to a non-zero value. If the second pathname begins in the server's domain but subsequently leaves it, the server returns FS_WOULD_BLOCK and sets name1ErrorP to zero. If the second pathname doesn't begin in the server's domain, it returns FS_CROSS_DOMAIN_OPERATION. At this point the client does a FS_GET_ATTR on the *parent* of the second pathname to make sure that further redirections do not lead the second pathname back to the server's domain. The name of the parent is easily computed by trimming off the last component of the second pathname. If the FS_GET_ATTR gets a redirect the client reiterates, otherwise the FS_RENAME (or FS_LINK) fails.

13. FS_MKDIR This is used to create a directory. The request data is a null terminated pathname. The request parameters are the Fs_OpenArgs described above for FS_OPEN, with the type equal to FS_DIRECTORY. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceeded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

14. FS_RMDIR This is used to remove a directory. The request data is a null terminated pathname. The request parameters are the Fs_LookupArgs described above for FS_UNLINK. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceeded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

15. FS_MKDEV This is used to create a device file. The request data is a null terminated pathname. The request parameters, Fs_MakeDeviceArgs, are defined below. If the return code is FS_LOOKUP_REDIRECT, then the reply data is a new pathname (again preceeded by 4 bytes of junk), and the reply parameters is the length of the embedded prefix.

```
typedef struct Fs_MakeDeviceArgs {
        Fs_OpenArgs open;
        Fs_Device device;
} Fs_MakeDeviceArgs;

typedef struct Fs_Device {
        int     serverID;
        int     type;
        int     unit;
        ClientData      data;
} Fs_Device;
```

The Fs_MakeDeviceArgs are a slight super-set of the Fs_OpenArgs. They additionally contain a Fs_Device structure that defines the server, type, and unit of the peripheral device. (The data field is not used in the RPC.) The serverID is a Sprite Host ID. If the special value FS_LOCALHOST_ID is used, then this device file always specifies the device attached to the local host. Otherwise, it specifies a devices at a particular host.

```
#define FS_LOCALHOST_ID     -1
```

Some of the device types are defined in Table D-7, although this list is not guaranteed to be complete. These types are defined in <kernel/dev/devTypes.h>. The unit number is device specific, and no attempt is made to specify them here. (For example, the ethernet device encodes the protocol number in the unit. The SCSI devices encode the controller number, target ID, and LUN.)

Device Types	
DEV_TERM	0
DEV_SYSLOG	1
DEV_SCSI_WORM	2
DEV_PLACEHOLDER_2	3
DEV_SCSI_DISK	4
DEV_SCSI_TAPE	5
DEV_MEMORY	6
DEV_XYLOGICS	7
DEV_NET	8
DEV_SCSI_HBA	9
DEV_RAID	10
DEV_DEBUG	11
DEV_MOUSE	12

Table D-7. Definitions for device types.

16. FS_LINK This creates another directory entry to an existing object. The two objects are restricted to be within the same file system domain. The request and reply messages have the same format as FS_RENAME. This is an operation on two pathnames, and the comments regarding pathname redirection from FS_RENAME apply.

17. FS_SYM_LINK This is defined but not yet supported. Instead, symbolic links and remote links are created by creating a file using FS_OPEN and type FS_SYMBOLIC_LINK or FS_REMOTE_LINK, and then writing the value of the link with FS_WRITE. This should change because it interacts poorly with systems that have a different format for their remote links. (For example, for no good reason Sprite includes a null character in its implementation of symbolic links, while UNIX does not. NFS access to Sprite are confused by this, and it is possible to create bad symbolic links on an NFS server via the Sprite NFS pseudo-file-system.)

18. FS_GET_ATTR This is used to get the attributes of the object behind an open I/O stream. The parameter area of the request contain a Fs_FileID, which has been defined above. This is the same as the ioFileID returned from an FS_OPEN. The request and reply data areas are empty. The reply parameter area contains a Fs_Attributes structure defined below.

```
typedef struct Fs_Attributes {
        int             serverID;
        int             domain;
        int             fileNumber;
        int             type;
        int             size;
        int             numLinks;
        unsigned int    permissions;
        int             uid;
        int             gid;
        int             devServerID;
        int             devType;
        int             devUnit;
        Time            createTime;
        Time            accessTime;
        Time            descModifyTime;
        Time            dataModifyTime;
        int             blocks;
        int             blockSize;
        int             version;
        int             userType;
        int             pad[4];
} Fs_Attributes;
```

19. FS_SET_ATTR This is used to set the attributes of an object behind an I/O stream. The request parameters contain the FsRemoteSetAttrParams structure, which is defined below. The request data area, reply data area, and reply parameter area are all empty.

```
typedef struct FsRemoteSetAttrParams {
        Fs_FileID       fileID;
        Fs_UserIDs      ids;
        Fs_Attributes   attrs;
        int             flags;
} FsRemoteSetAttrParams;
```

The fileID identfies the object, and was returned as the ioFileID from the FS_OPEN RPC. The Fs_UserIDs structure is needed to verify that the attributes can be changed, and this has been defined above with FS_OPEN. The flags field indicates which attributes are to be set. These flags are described in Table D-8.

20. FS_GET_ATTR_PATH This is used to get the attributes of a file system object from the file server. This is an operation on a pathname, so the request data area contains a null terminated pathname. The request parameters are the Fs_OpenArgs previously described. The reply parameters are a Fs_GetAttrResultsParam structure, which is defined below. If the return code is FS_LOOKUP_REDIRECT, then the reply data area

Set Attribute Flags		
Value	Name	Description
0x1F	FS_SET_ALL_ATTRS	Set all the attributes possible. See the following definitions.
0x01	FS_SET_TIMES	Set the access and modify times. Used to implement UNIX utimes().
0x02	FS_SET_MODE	Set the permissions. Used to implement UNIX chmod().
0x04	FS_SET_OWNER	Set the owner and group. If either of the gid and uid fields in the Fs_Attributes structure are -1, then they are not changed.
0x08	FS_SET_FILE_TYPE	Set the user-defined file type. The user-defined types currently in use are defined in the next table.
0x10	FS_SET_DEVICE	Set the device attributes. Used to implement Fs_MakeDevice (UNIX mknod()).

Table D-8. Values for the flags field in FsRemoteSetAttrParams.

User-defined File Types		
FS_USER_TYPE_UNDEFINED	0	Not used.
FS_USER_TYPE_TMP	1	Temporary file.
FS_USER_TYPE_SWAP	2	VM swap file.
FS_USER_TYPE_OBJECT	3	Program object file.
FS_USER_TYPE_BINARY	4	Complete program image.
FS_USER_TYPE_OTHER	5	Everything else.

Table D-9. Values for the user-defined file type attribute.

contains a returned pathname. NB: in this case there is *no* 4 bytes of padding in the data area! (Sorry for this ugly inconsistency.)

```
typedef union Fs_GetAttrResultsParam {
        int       prefixLength;
        struct AttrResults {
        Fs_FileID       fileID;
        Fs_Attributes  attrs;
        } attrResults;
} Fs_GetAttrResultsParam;
```

The prefixLength is returned with the FS_LOOKUP_REDIRECT status. Otherwise, the fileID is the same as the ioFileID returned from an FS_OPEN, and this is used to do a FS_GET_IO_ATTR RPC with the I/O server. The Fs_Attributes structure has been defined above.

21. FS_SET_ATTR_PATH This is used to set the attributes of a named file system object. The request data area contains a null terminated pathname. The request parameters are defined below. The reply data area contains a new pathname if the return code is FS_LOOKUP_REDIRECT. (No padding bytes.) The reply data area contains a Fs_GetAttrResultsParam structure which has been defined above. If FS_LOOKUP_REDIRECT is returned then only the prefixLength is defined in the Fs_GetAttrResultsParam.

```
typedef struct Fs_SetAttrArgs {
        Fs_OpenArgs  openArgs;
        Fs_Attributes  attr;
        int       flags;
} Fs_SetAttrArgs;
```

Each of these fields have been described above. The flags define which attributes to set. See FS_SET_ATTR for a description.

22. FS_GET_IO_ATTR This is used to get the attributes that are maintained by the I/O server, typically the access and modify times. The general approach to getting attributes is to first contact the file server (with either FS_GET_ATTR or FS_GET_ATTR_PATH) to get the initial version of the attributes, and then use this RPC to get the most up-to-date access and modify times. This is only applicable to remote devices and remote pseudo-devices.

The request parameter area contains a Fs_GetAttrResultsParam that has been initialized by a FS_GET_ATTR or FS_GET_ATTR_PATH RPC. The reply parameter area contains a Fs_Attributes structure. The request and reply data areas are empty.

23. FS_SET_IO_ATTR This is used to update the attributes that are maintained by the I/O server, typically the access and modify times. Setting attributes is structured the same as with getting them, so the file server is contacted first (with FS_SET_ATTR or FS_SET_ATTR_PATH). The complete attributes are returned from these calls, and then used as the request parameters in this RPC.

The request message contains a FsRemoteSetAttrParams structure, which is defined below. The request data area, the reply parameter area, and the reply data area are empty.

```
typedef struct FsRemoteSetAttrParams {
        Fs_FileID       fileID;
        Fs_UserIDs      ids;
        Fs_Attributes   attrs;
        int     flags;
} FsRemoteSetAttrParams;
```

The fileID identifies the object, and is the same as the ioFileID returned from an FS_OPEN RPC. The Fs_UserIDs and Fs_Attributes have been described above. The flags indicate what attributes to set, and these are indicated above as well.

24. FS_DEV_OPEN This is used to complete an Fs_Open (UNIX open()) of a remote device or remote pseudo-device. This is done after an FS_OPEN RPC has returned an ioFileID with a type of FSIO_RMT_DEVICE_STREAM or FSIO_RMT_PDEV_STREAM. The request parameter area contains a FsDeviceRemoteOpenParam structure, which is defined below. The reply parameter area contains a new ioFileID so the I/O server can modify this if it wants to. The request and reply data areas are empty.

```
typedef struct FsDeviceRemoteOpenParam {
        Fs_FileID       fileID;
        int     useFlags;
        int     dataSize;
        FsrmtUnionData          openData;
} FsDeviceRemoteOpenParam;
```

The fileID is the ioFileID returned from FS_OPEN. The useFlags are the newUseFlags returned from FS_OPEN. The dataSize indicates the number of valid bytes in openData.

The FsrmtUnionData has been described previously.

25. FS_SELECT This RPC is used to poll a remote I/O server to determine if an object is ready for I/O. The input parameters include a fileID that identifies the object, three fields that correspond to the read, write, and execute state of the object, and process information used for remote waiting. If a field is non-zero it means an application is querying that state. The return parameters contains copies of these fields, and the I/O server should clear a field to zero if that state is not applicable. In this case, the I/O server should save the Sync_RemoteWaiter information so it can notify the waiting process when the objects state changes.

```
typedef struct FsRemoteSelectParams {
        Fs_FileID      fileID;
        int      read;
        int      write;
        int      except;
        Sync_RemoteWaiter waiter;
} FsRemoteSelectParams;

typedef struct FsRemoteSelectResults {
        int      read;
        int      write;
        int      except;
} FsRemoteSelectResults;
```

NB: The read, write, and except fields are defined to be zero or non-zero, and the fields in the result should be exact copies of the request fields, or they should be reset to zero. (If non-zero they happen to be the bit that corresponds to the bit in the select mask. It is only appropriate to copy the bit or clear it. Don't blindly set the reply fields to 1.)

26. FS_IO_CONTROL This is used to do an object-specific operation. A number of generic I/O controls are defined below, and the implementation of different objects are free to define more I/O controls. The request parameter area contains a FsrmtIOCParam structure, which is defined below. The reply parameter area contains a Fs_IOReply structure that defines the amount of data returned, and a signal to generate, if any. The request and reply data areas contain data blocks that are uninterpreted by generic kernel code. In particular, they cannot be byteswapped except by the implementation of the I/O control handler. The parameters include a byteOrder field so the handler can detect a mismatch. In this case it should byteswap in the request data block so it can properly interpret it, and also byteswap the reply datablock so the application process on the remote client can properly interpret it.

```
typedef struct FsrmtIOCParam {
        Fs_FileID      fileID;
        Fs_FileID      streamID;
        Proc_PID       procID;
```

```
    Proc_PID    familyID;
        int     command;
        int     inBufSize;
        int     outBufSize;
        Fmt_Format  format;
        int     uid;
} FsrmtIOCParam;
```

The fileID and streamID have been returned by FS_OPEN. The procID and familyID identify the process and its process group. The sizes indicate how much data is being sent to the I/O server and the maximum amount that can be accepted in return. The format defines a byte ordering, and it is used in conjunction with the Format library package to do byteswapping. The uid specifies the user ID of the application process. The command is the I/O control operation, and the generic ones are defined below.

1 IOC_REPOSITION

Reposition the current access position of the I/O stream. The input data block contains the following structure to define the new access position.

```
#define IOC_BASE_ZERO       0
#define IOC_BASE_CURRENT    1
#define IOC_BASE_EOF        2

typedef struct Ioc_RepositionArgs {
        int base;
        int offset;
} Ioc_RepositionArgs;
```

2 IOC_GET_FLAGS

Return the flag bits associated with an I/O stream. The reply data block contains an integer with the flag bits. The following bits are defined for all objects, although other objects may define more flags.

```
#define IOC_GENERIC_FLAGS 0xFF
#defineIOC_APPEND   0x01
#define IOC_NON_BLOCKING  0x02
#define IOC_ASYNCHRONOUS0x04
#define IOC_CLOSE_ON_EXEC0x08
```

3 IOC_SET_FLAGS

Set the flags on an I/O stream. The request data block contains an integer which is to be new version of the flag word. It completely replaces the old flag word.

4 IOC_SET_BITS

Set individual bits in the flags word of an I/O stream. The request data block contains an integer with the desired bits set.

5 IOC_CLEAR_BITS

Clear individual bits in the flags word of an I/O stream. The request data block contains an integer with the desired bits set.

6 IOC_TRUNCATE

Truncate an object to a specified length. The input data block contains an integer which is the desired length.

7 IOC_LOCK

Place an advisory lock on an object. Used to implement UNIX flock(). The request data block contains the following structure. If the lock cannot be obtained then FS_WOULD_BLOCK should be returned and the process information should be saved at the I/O server for a subsequent REMOTE_WAKEUP RPC back to the client.

```
typedef struct Ioc_LockArgs {
        int       flags;
        int       hostID;
        Proc_PID      pid;
        int       token;
} Ioc_LockArgs;

#define IOC_LOCK_SHARED    0x1
#define IOC_LOCK_EXCLUSIVE         0x2
#define IOC_LOCK_NO_BLOCK          0x8
```

The flag bits are defined above, and specify if the lock should be exclusive, in which case it is blocked by either an existing exclusive lock or by a shared lock, or whether it is a shared lock, in which case it can co-exist with other shared locks but not an exclusive lock. If IOC_LOCK_NO_BLOCK is set then the application process will not be blocked by the client in the case of FS_WOULD_BLOCK, so the I/O server doesn't have to remember the process information.

8 IOC_UNLOCK

Remove an advisory lock on an object. The complement of IOC_LOCK.

9 IOC_NUM_READABLE

Returns the number of bytes available on an I/O stream. The input data block contains the current offset of the stream. The return data block contains the number of bytes available on the stream.

10 IOC_GET_OWNER

Returns the owner of an I/O stream, which is either a process or a process group. The return data block contains the following structure, along with definitions for the procOrFamily field.

```
#define IOC_OWNER_FAMILY 0x1
#define IOC_OWNER_PROC     0x2

typedef struct Ioc_Owner {
        Proc_PID      id;
        int       procOrFamily;
} Ioc_Owner;
```

11 IOC_SET_OWNER

This defines the owner of an I/O stream. The request data block contains the

Ioc_Owner structure.

12 IOC_MAP
Obsolete, superceeded by IOC_MMAP_INFO.

13 IOC_PREFIX
This returns the prefix under which a stream was opened. This is used to implement the getwd() library call. The reply data block contains the prefix, which is null terminated.

14 IOC_WRITE_BACK
This is used to write-back a range of bytes of a file to the server's disk. The cache will align the write-back on block boundaries that include the specified range of bytes. The request data area contains the following structure.

```
typedef struct Ioc_WriteBackArgs {
        int     firstByte;
        int     lastByte;
        Boolean         shouldBlock;
} Ioc_WriteBackArgs;
```

15 IOC_MMAP_INFO
Tell the I/O server that a client is mapping a stream into memory. The request data area contains the following structure. The isMapped field is 0 if the client is unmapping the stream, and 1 if it is mapping the stream.

```
typedef struct Ioc_MmapInfoArgs {
        int     isMapped;
        int     clientID;
} Ioc_MmapInfoArgs;
```

((1<<16)-1) IOC_GENERIC_LIMIT
The Sprite kernel reserves the numbers below this for generic I/O control commands. Other device drivers and pseudo-device servers define their own I/O controls. Look at the README file ‘‘/sprite/src/include/dev/README’’ for details.

27. FS_CONSIST This is issued by a file server as a side effect of an FS_OPEN RPC. It is a command to a client (not the one doing the FS_OPEN) to control its cache so that future accesses see consistent data. The request parameter area contains the following structure, and the request data area, reply data area, and reply parameter area are empty. The client should respond immediately to this RPC and perform the cache consistency actions in the background. The client issues a FS_CONSIST_REPLY RPC to the server when it has completed the requested actions. This is a crude way of doing parallel RPCs to many clients. The server sets up a short timeout (about 1 minute) for the client to complete its actions, and it will let an open complete anyway if this timeout expires and a rogue client has not responded to a consistency request.

```
typedef struct ConsistMsg {
        Fs_FileID       fileID;
        int     flags;
```

```
        int    openTimeStamp;
               int       version;
    } ConsistMsg;
```

The fileID identifies the file, and it is the same as the ioFileID returned from the FS_OPEN RPC. The flags are explained below, and they indicate what action the client should take. The openTimeStamp is the time stamp that the server thinks corresponds to the last openTimeStamp it returned to the client. The version should match with the last version returned to the client in the Fsio_FileState. (It will eventually replace the openTimeStamp altogether.) The point of the openTimeStamp is that if two clients open the same file at the same time, then the reply to one client's FS_OPEN may loose a race with a FS_CONSIST RPC generated by the second client's open. If a client receives a FS_CONSIST RPC with an openTimeStamp ''in the future'' it drops the consistency request and returns FAILURE (1). This forces the file server to retry the FS_CONSIST call, giving the reply to the FS_OPEN a chance to arrive at the client. The consistency actions are defined below.

0x01 FSCONSIST_WRITE_BACK_BLOCKS
: The client should write back any dirty blocks that are lingering in its cache.

0x02 FSCONSIST_INVALIDATE_BLOCKS
: The client should stop caching the file because it is now concurrently write shared by different hosts. All future I/O operations on this file should bypass the client cache and go through to the file server.

0x04 FSCONSIST_DELETE_FILE
: This is issued as a side effect of a FS_REMOVE RPC if the client has dirty blocks for the file. This is done even if it is the same client as the one currently making the FS_REMOVE RPC.

0x08 FSCONSIST_CANT_CACHE_NAMED_PIPE
: This is reserved for if we ever re-implement named pipes.

0x10 FSCONSIST_WRITE_BACK_ATTRS
: The client should write-back its notion of the access and modify times of the file that it is caching. This is generated as a side effect of FS_GET_ATTR and FS_GET_ATTR_PATH RPCs by other clients. This is only done if the client is actively using the file, and it is suppressed if the client only has the file open for execution. The attributes are returned with the FS_CONSIST_REPLY RPC.

28. FS_CONSIST_REPLY This is issued by the client when it has completed the consistency actions requested by the server. The request parameter area contains the following structure.

```
    typedef struct ConsistReply {
        Fs_FileID        fileID;
        Fscache_Attributes     cachedAttr;
        ReturnStatus    status;
    } ConsistReply;
```

The fileID indicates the file that was acted on. The client always returns its notion of the attributes of the file because it updates these while caching the file. The status indicates whether it could comply with the request. If the server's disk is so full that a write-back could not be made then this status is FS_DISK_FULL. The data is not lost, but it lingers in the client's cache until the write-back can succeed. However, this means that an open() can fail with a disk full error!

29. FS_COPY_BLOCK This is used during fork() to copy a swap file on the file server. This prevents the client from reading a swap file over the network just to copy it and write it back. The request parameter area contains the folowing structure, and the request data area, reply parameter area, and reply data area are empty.

```
typedef struct FsrmtBlockCopyParam {
        Fs_FileID    srcFileID;
        Fs_FileID    destFileID;
        int     blockNum;
} FsrmtBlockCopyParam;
```

The srcFileID and destFileID have been returned from FS_OPEN when the swap files were opened. The blockNum specifies the FS_BLOCK_SIZE (4096 bytes) block to copy. This is a logical block number because the client has no notion of where the swap files live on disk.

30. FS_MIGRATE This is used during process migration to inform the I/O server that an I/O stream has migrated to a new client. This is invoked from the destination client as part of creating the process. The request parameter area contains the following structure.

```
typedef struct FsMigInfo {
        Fs_FileID    streamID;
        Fs_FileID    ioFileID;
        Fs_FileID    nameID;
        Fs_FileID    rootID;
        int     srcClientID;
        int     offset;
        int     flags;
} FsMigInfo;
```

This information is packaged up on the source client when the process migrates away. The streamID, ioFileID, and nameID are those that have been returned from a previous FS_OPEN. The rootID was specified in the FS_OPEN request that created the stream. The srcClientID is the client were the process left. The offset is the offset at the time the process left. (This is a bug, the offset should be returned from the source client as part of the FS_RELEASE RPC.) The flags are the flags from the stream.

The reply parameter area of the FS_MIGRATE RPC contains the following structure. The request and reply data areas are empty.

```
typedef struct FsrmtMigParam {
        int     dataSize;
```

```
    FsrmtUnionData     data;
        FsrmtMigrateReply     migReply;
} FsrmtMigParam;

typedef struct FsrmtMigrateReply {
        int flags;
        int offset;
} FsrmtMigrateReply;

#define FS_RMT_SHARED        0x04000000
```

The I/O server returns the same FsrmtUnionData as it does in an FS_OPEN RPC, and the client uses this to set up its I/O stream data structures. The I/O server also tells the client the new stream offset to use, and it gives the client a new version of the stream flags. If the flags in include the FS_RMT_SHARED bit then processes on different hosts are sharing the stream. In this case the offset in the clients' stream descriptors are not valid, and I/O operations on the object have to go through to the I/O server. The I/O server keeps a shadow stream descriptor that contains the valid stream offset in this case.

31. FS_RELEASE This is used by the I/O server during process migration to tell the source of a migrated process that it can release an I/O stream that had been associated with the process. Recall that fork() and dup() create extra references to a stream descriptor, so this call is used to release that reference. This cannot be done safely at the time the process leaves, so it is done as a side effect of the FS_MIGRATE RPC issued from the destination client. At this time the current offset in the source client's stream descriptor is also returned to the I/O server, in case it needs to be cached there while the stream is shared by processes on different hosts. The request parameter area contains the ID of the stream that migrated, and the reply contains an inUse flag and the current offset. The inUse flag should be set if their are still processes on the source client that reference the stream descriptor.

```
typedef struct {
        Fs_FileID streamID;
} FsStreamReleaseParam;

typedef struct {
        Boolean        inUse;
        int        offset;
} FsStreamReleaseReply;
```

32. FS_REOPEN This is used during the state recovery protocol to inform the I/O server about I/O streams in use by a client. The request and reply parameter areas vary depending on the object being reopened. The following structures are possible, although note that every request structure contains a Fs_FileID as its first element. Also, the client should map its FSIO_RMT stream types to the corresponding FSIO_LCL stream types before making the RPC.

```
typedef struct FsRmtDeviceReopenParams {
        Fs_FileID       fileID;
        Fsutil_UseCounts use;
} FsRmtDeviceReopenParams;

typedef struct Fsutil_UseCounts {
        int     ref;
        int     write;
        int     exec;
} Fsutil_UseCounts;
```

The reopen parameters identify the object and specify how many streams the client has to it. Note that the ref field in Fsutil_UseCounts is not the same as the number of reading streams, but it is the total number of streams. This is a mistake and will be fixed eventually; it makes it impossible for a reader to reopen ‘‘/dev/syslog’’, which is a single reader/multiple writer device. The request data area, the reply parameter area, and the reply data area are empty in the case of device reopening.

```
typedef struct Fsio_PipeReopenParams {
        Fs_FileID       fileID;
        Fsutil_UseCounts use;
} Fsio_PipeReopenParams;
```

A pipe may become remote due to process migration, therefore it may have to be reopened if the client looses touch with the server. If the server can crashed then the reopen will fail, but if there has only been a network partition the reopen may succeed. The request data area, the reply parameter area, and the reply data area are empty in the case of pipe reopening.

```
typedef struct Fsio_FileReopenParams {
        Fs_FileID       fileID;
        Fs_FileID       prefixFileID;
        Fsutil_UseCounts        use;
        Boolean         flags;
        int     version;
} Fsio_FileReopenParams;

#define FSIO_HAVE_BLOCKS 0x1
#define FS_SWAP         0x4000
```

The reopen parameters for a file specify the file and the prefix of the domain of the file. This is needed to validate that the server still has the disk mounted. The Fsutil_UseCounts are as described above. The flags include FSIO_HAVE_BLOCKS, FS_SWAP, and FS_MAP to indicate if the client has dirty data blocks, is using the file for VM backing store, or is mapping the file into its memory. The version number is the version that the client has cached. The reply parameters when reopening a file are the Fsio_FileState described above. The client should verify that the version number it has is correct, just as it does during an FS_OPEN.

```
typedef struct FspdevControlReopenParams {
        Fs_FileID   fileID;
        int     serverID;
        int     seed;
} FspdevControlReopenParams;
```

The file server that stores a pseudo-device file also keeps some state as to whether there is currently a server process for the pseudo-device so it can prevent conflicts. If the file server crashes this information has to be restored, and it is done by reopening the pseudo-device control handle. The fileID in the reopen parameters is that returned from FS_OPEN when the server process opened the pseudo-device with the FS_PDEV_MASTER flag. The serverID is -1 if the server process has gone away since the file server crashed. The seed is used by the file server to generate unique fileIDs for the connections to the pseudo-device, and this needs to be restored, too. Under normal operation the file server increments its seed everytime a new open is done on the pseudo-device, and it puts the seed into the low-order 12 bits of the minor field in the ioFileID returned from FS_OPEN. The I/O server knows about this, and it extracts the seed from the ioFileID so it can restore it during recovery.

```
typedef struct StreamReopenParams {
        Fs_FileID   streamID;
        Fs_FileID   ioFileID;
        int     useFlags;
        int     offset;
} StreamReopenParams;
```

After all the other kinds of I/O handles have been reopened at a server, the client reopens its stream descriptors that reference the I/O handles. The reopen specifies the streamID and the ioFileID, so the I/O server can verify that its shadow stream descriptor connects to the same I/O handle that the client thinks it should. The offset is used to recover the offset in the server's shadow stream descriptor. (This isn't implemented. If the file server crashes while a stream is shared by processes on different hosts, then the shared offset is lost. This needs to be fixed, perhaps by adding an offset to the Fs_IOReply structure so the client can cache the offset.)

33. FS_RECOVERY This is used after a client has completed its state recovery. This is needed because the server drops regular FS_OPEN RPCs while a client is doing FS_REOPEN RPCs. Specifically, after a server gets a FS_REOPEN from a client it drops an FS_OPEN (from that client) until it recieves an FS_RECOVERY RPC. The FS_RECOVERY RPC can also be used by a client to signal that it is begining the recovery protocol, but this is not necessary. The parameter area of the request message contains a single integer that contains a flag CLT_RECOV_IN_PROGRESS if the client is initiating recovery, and that has no flag (zero value) when the client completes recovery.

```
#define CLT_RECOV_IN_PROGRESS   0x1
#define CLT_RECOV_COMPLETE      0x0
```

34. FS_DOMAIN_INFO This is used to get information about a file system domain, including the amount of disk space available. The request parameter area contains the fileID associated with the prefix of the domain. This is returned from the FS_PREFIX RPC. The reply parameter area contains a FsDomainInfoResults structure. The request and reply data areas are empty.

```
typedef struct FsDomainInfoResults {
        Fs_DomainInfo        domain;
        Fs_FileID       fileID;
} FsDomainInfoResults;

typedef struct {
        int     maxKbytes;
        int     freeKbytes;
        int     maxFileDesc;
        int     freeFileDesc;
        int     blockSize;
        int     optSize;
} Fs_DomainInfo;
```

The fileID that is returned is the user-visible fileID that an application program would see if it did a GET_ATTR_PATH on the prefix. With a pseudo-file-system this is different than the internal fileID associated with the prefix, which identfies a request-response connection between the kernel and the server process. The Fs_DomainInfo indicates the maximum size of the file system, the number of free kilobytes, the maximum number of file descriptors, the number of free descriptors, the native blocksize of the file system, and the optimal transfer size of the file system.

35. PROC_MIG_COMMAND This is used to transfer process state between Sprite hosts. The request parameter area contains a process ID and a command identifier. The request data area contains command specific data. The reply parameter area contains a return status, and optionally some command specific data. The migration commands are described below, along with their command specific data.

```
typedef struct {
        Proc_PID        remotePid;
        int             command;
} ProcMigCmd;
```

0 PROC_MIGRATE_CMD_INIT

This is used to request permission to migrate to another host. The remotePid field of the ProcMigCmd is NIL if the process is leaving its home node. During eviction, when a process is migrating back home, the remotePid field is the home node process ID. The request data area contains a ProcMigInitiateCmd structure, and the reply parameter area contains the processID for the process on the remote host.

```
typedef struct {
        int     version;
```

```
        Proc_PID    processID;
        int         userID;
        int         clientID;
    } ProcMigInitiateCmd;
```

The version is a process migration implementation version number to ensure that the two hosts are compatible. The processID is the ID of the process that wishes to migrate. The userID is that of the owner of the process. The clientID is the Sprite hostID of the host issuing the request.

1 PROC_MIGRATE_CMD_ENTIRE

This transfers the process control block. The request data area contains an encapsulated control block. The exact format of the encapsulated control block is machine specific and will not be described here. The reply data area is empty.

2 PROC_MIGRATE_CMD_UPDATE

This is used to update the state of a migrated process. The request data area contains an UpdateEncapState structure, which contains the few fields of a Sprite process control block that a process can modify.

```
    typedef struct {
        int     familyID;
        int     userID;
        int     effectiveUserID;
        int     billingRate;
    } UpdateEncapState;
```

3 PROC_MIGRATE_CMD_CALLBACK

Not used.

4 PROC_MIGRATE_CMD_DESTROY

This is called to kill a migrated process. The request and reply data areas are empty.

5 PROC_MIGRATE_CMD_RESUME

This is called to continue execution of a suspended migrated process. The request and reply data areas are empty.

6 PROC_MIGRATE_CMD_SUSPEND

This is called to suspend execution of a migrated process. The request and reply data areas are empty.

36. PROC_REMOTE_CALL This is used to forward a system call from a migrated process back to its home node. Most system calls are not fowarded, only a few that depend on state maintained at the home node. The format of the request and reply are implemnetation and system call specific, and are not described here.

37. PROC_REMOTE_WAIT This is used when a migrated process waits for child processes. Communication with the home node is required because synchronization with process creation, process exit, and waiting is done there. The parameter area contains a ProcRemoteWaitCmd structure, and the request data area contains an array of processIDs on which to wait. The reply parameter area is empty and the reply data area contains a ProcChildInfo structure. (Byte ordering isn't an issue in the data area because process migration only works between hosts of the same machine architecture.)

```
typedef struct {
        Proc_PID      pid;
        int           numPids;
        Boolean       flags;
        int           token;
} ProcRemoteWaitCmd;

typedef struct {
        Proc_PID      processID;
        int           termReason;
        int           termStatus;
        int           termCode;
        int           numQuantumEnds;
        int           numWaitEvents;
        Timer_Ticks   kernelCpuUsage;
        Timer_Ticks   userCpuUsage;
        Timer_Ticks   childKernelCpuUsage;
        Timer_Ticks   childUserCpuUsage;
} ProcChildInfo;
```

38. PROC_GETPCB This is used to return the process control block of a migrated process for implementation of the **ps** (process status) application program. The request parameter area contains an integer with value GET_PCB (0x1) or GET_SEG_INFO (0x2). With GET_PCB the request data area contains the processID (also an integer). The reply parameter area contains a Proc_PCBInfo structure, and the reply data area contains the argument string of the process. The Proc_PCBInfo is described in ''/sprite/lib/include/proc.h''. With GET_SEG_INFO the request data area contains a virtual memory segment number, and the reply parameter area contains a Vm_SegmentInfo structure, which is described in ''/sprite/lib/include/vm.h''.

39. REMOTE_WAKEUP This is used to notify a remote process that some event has occurred. The process has presumably registered itself via some blocking call such as FS_READ or FS_WRITE, whose parameters include a Sync_RemoteWaiter structure. The request parameter area of REMOTE_WAKEUP contains a Sync_RemoteWaiter structure, and the request data area, reply parameter area, and reply data area are empty. Note that this wakeup message can race with the process's decision to wait at the other host. To foil the race condition a process must be marked as in the process of deciding to wait. In the Sprite implementation, this is done by clearing a *notify* bit kept in the process's control block. When a REMOTE_WAKEUP RPC is recieved by a Sprite host, the notify bit in the process control block is set. Before actually blocking a process (in response to a FS_WOULD_BLOCK return code) the Sprite kernel checks that the notify bit has not been set asynchronously via this RPC. If the bit has been set, then the process is not blocked and it retries its operation immediately. If this technique were not used then notifications might get lost and hang the process.

40. SIG_SEND This is used to issue a signal to a remote process. The request parameter are contains a SigParams structure. The request data area, reply parameter area, and reply data area are empty.

```
typedef struct {
        int         sigNum;
        int         code;
        Proc_PID    id;
        Boolean     familyID;
        int         effUid;
} SigParms;
```

The sigNum is a Sprite signal, and the code field is used to modify this. The id is a process identifier if the familyID field is zero, otherwise id is a process group identifier. The effUid is the effective user ID of the signaling process, and this is used to verify permissions.

41. Bugs and Ommisions

This specification is based on the Sprite implementation as of Fall 1989. There are a couple of known bugs in it, and it is a bit crufty. However, there is a lot of inertia behind the network interface because changing it requires coordinated changes on all Sprite hosts. Future changes to the interface will ideally be backward compatible with this interface by introducing new RPCs that fix certain bugs, while retaining the original for compatibility with hosts running older versions of Sprite. The known bugs in the interface are summarized below.

41.1. FS_REMOTE_LINK vs. FS_SYMBOLIC_LINK The Fs_ReadLink (or UNIX readlink) system call is implemented as an FS_OPEN followed by an FS_READ. The type field in the Fs_OpenArgs is specified as FS_REMOTE_LINK in the current implementation, but the server should also allow regular symbolic links to be opened.

41.2. Device Attributes Currently, while the I/O server maintains the access and modify times for a device while it is opened, this information is not pushed back to the file server when the device is closed.

41.3. Pathname Redirection There is some cruft in the way pathnames are returned from the server. In some RPCs there is an extra 4 bytes in the data area that preceeds the pathname, but in the Attributes RPCs the padding is gone. This is a hold-over from pre-byteswapping days when the 4 bytes in the data area contained the prefix length. Similarly, with the FS_RENAME and FS_LINK, there are 8 bytes of junk before the returned pathname.

41.4. FS_SERVER_WRITE_THRU This flag is currently private to the client side of the implementation. It could be passed through to the server to force a write-through to disk. Currently, however, the client and server writing policies are completely independent. Ordinarily clients uses a 30 second delay, and servers use write-back-ASAP. This means that a file ages in the client'cache for 30 seconds, and then gets scheduled for a disk write-back after the last block arrives from the client. Note that a client can use fsync(), in which case the blocks are forced through to the servers disk, and fsync() doesn't return until after that has happened.

References

Accetta86. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, ‘‘Mach: A New Kernel Foundation for UNIX Development.’’, *Proc. of Summer USENIX*, July 1986.

Atlas86. A. Atlas and P. Flinn, ‘‘Error Recovery in a Stateful Remote Filesystem’’, *USENIX Conference Proceedings*, Summer 1986, 355-365.

Back87. J. J. Back, M. W. Luppi, A. S. Melamed and K. Yueh, ‘‘A Remote-File Cache for RFS’’, *USENIX Conference Proceedings*, Summer 1987, 273-279.

Bartlett81. J. Bartlett, ‘‘A NonStop (tm) Kernel’’, *Proc. of the 8th Symp. on Operating System Prin.*, Dec. 1981, 22-29.

Bershad88. B. N. Bershad and C. B. Pinkerton, ‘‘Watchdogs - Extending the UNIX File System’’, *USENIX Association 1988 Winter Conference Proceedings*, Feb. 1988, 267-275.

Bershad89. B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, ‘‘Lightweight Remote Procedure Call’’, *Proc. 12th Symp. on Operating System Prin., Operating Systems Review 23*, 5 (December 1989), 102-113.

Birrell82. A. D. Birrell, ‘‘Grapevine: An Exercise in Distributed Computing’’, *Comm. of the ACM 25*, 4 (Apr. 1982), 260-274..

Birrell84. A. D. Birrell and B. J. Nelson, ‘‘Implementing Remote Procedure Calls’’, *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

Brownbridge82. D. R. Brownbridge, L. F. Marshall and B. Randell, ‘‘The Newcastle Connection or UNIXes of the World Unite!’’, *Software Practice and Experience 12* (1982), 1147-1162.

Burrows88. M. Burrows, ‘‘Efficient Data Sharing’’, PhD Thesis, Dec. 1988. University of Cambridge, Computer Laboratory.

Cabrera87. L. F. Cabrera and J. Wyllie, ‘‘QuickSilver Distributed File Services: An Architecture for Horizontal Growth’’, Research Report 5578 (56697) 4/1/87, IBM Almaden Research Center, Apr. 1987.

Chartock87. H. Chartock, ‘‘RFS in SunOS’’, *USENIX Conference Proceedings*, Summer 1987, 281-290.

Cheriton84. D. R. Cheriton, ‘‘The V Kernel: A software base for distributed systems.’’, *IEEE Software 1*, 2 (Apr. 1984), 19-42.

Cheriton87. D. R. Cheriton, ‘‘UIO: A uniform I/O interface for distributed systems’’, *ACM Trans. on Computer Systems 5*, 1 (Feb. 1987), 12-46.

Cheriton89. D. R. Cheriton and T. P. Mann, ‘‘Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance’’, *Trans.*

Computer Systems 7, 2 (May 1989), 147-183.

Clark85. D. Clark, ‘‘The Structuring of Systems Using Upcalls’’, *Proc. 10th Symp. on Operating System Prin., Operating Systems Review 19*, 5 (December 1985), 171-180.

Douglis87. F. Douglis, ‘‘Process Migration in Sprite’’, Technical Report UCB/Computer Science Dpt. 87/343, University of California, Berkeley, Feb. 1987.

Douglis90. F. Douglis, ‘‘Transparent Process Migration for Personal Workstations’’, PhD Thesis, *in preparation* 1990. University of California, Berkeley.

Ellis83. C. S. Ellis and R. A. Floyd, ‘‘The Roe File System’’, *Proc. of the 3rd Symp. on Reliability in Distributed Software and Database Systems*, 1983, 175-181.

Fabry74. R. Fabry, ‘‘Capability-Based Addressing’’, *Comm. of the ACM 17*, 7 (July 1974), 403-412.

Feirtag71. R. J. Feirtag and E. I. Organick, ‘‘The Multics Input/Output System’’, *Proc. of the 3rd Symp. on Operating System Prin.*, 1971, 35-41.

Fitzgerald85. R. Fitzgerald and R. Rashid, ‘‘The Integration of Virtual Memory Management and Interprocess Communication in Accent’’, *Proc. 10th Symp. on Operating System Prin., Operating Systems Review 19*, 5 (December 1985), 13-14.

Goldberg74. R. Goldberg, ‘‘Survey of Virtual Machine Research’’, *IEEE Computer*, June 1974, 34-45.

Gray89. C. G. Gray and D. R. Cheriton, ‘‘Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency’’, *Proc. 12th Symp. on Operating System Prin., Operating Systems Review 23,* 5 (December 1989), 202-210.

Hagmann87. R. Hagmann, ‘‘Reimplementing the Cedar File System Using Logging and Group Commit’’, *Proc. of the 11th Symp. on Operating System Prin.*, Nov. 1987, 155-162.

Hill86. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, ‘‘Design Decisions in SPUR’’, *IEEE Computer 19*, 11 (November 1986).

Hisgen89. A. Hisgen, A. Birrell, T. Mann, M. Schroeder and G. Swart, ‘‘Availability and Consistency Tradeoffs in the Echo Distributed File System’’, *Proc. Second Workshop on Workstation Operating Systems*, Sep. 1989, 49-54.

Howard88. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham and M. J. West, ‘‘Scale and Performance in a Distributed File System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 51-81.

Hughes86. R. P. Hughes, ‘‘The Transparent Remote File System’’, *USENIX Conference Proceedings*, Summer 1986, 306-317.

Kazar89. M. Kazar, ‘‘Ubik: Replicated Servers Made Easy’’, *Proc. Second Workshop on Workstation Operating Systems*, Sep. 1989, 60-67.

Kleiman86. S. Kleiman, ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX’’, *USENIX Conference Proceedings*, June 1986, 238-247.

Lampson86. B. Lampson, ‘‘Designing a Global Name Service’’, *Proc. ACM SIGACT News-SIGOPS 5th Symposium on the Principals of Distributed Computing*, Aug. 1986, 1-10.

Leach82. P. J. Leach, B. L. Stumpf, J. A. Hamilton and P. H. Levine, ‘‘UIDS as Internal Names in a Distriubted File System’’, *Proc. of ACM SIGACT News-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1982, 34-41.

Leach83. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, ‘‘The Architecture of an Integrated Local Network’’, *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (Nov. 1983), 842-857.

Levine85. P. H. Levine, P. J. Leach and N. W. Mishkin, ‘‘Distributed File Systems for Intimate Sharing’’, *Proc. 1rst Int’l Conf. on Computer Workstations*, 1985, 282-285.

Lobelle85. M. C. Lobelle, ‘‘Integration of Diskless Workstations in UNIX United’’, *Software Practice and Experience 15*, 10 (Oct. 1985), 997-1010.

McKusick84. M. K. McKusick, ‘‘A Fast File System for UNIX’’, *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181-197..

Mitchell82. J. G. Mitchell and J. Dion., ‘‘A Comparison of Two Network-Based File Servers’’, *Comm. of the ACM 25*, 4 (Apr. 1982), 233-245..

Needham79. R. Needham, ‘‘System Aspects of the Cambridge Ring’’, *Proc. of the 7th Symp. on Operating System Prin.*, 1979, 82-85.

Nelson88a. M. Nelson, B. Welch and J. Ousterhout, ‘‘Caching in the Sprite Network File System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 134-154.

Nelson88b. M. N. Nelson, ‘‘Physical Memory Management in a Network Operating System’’, PhD Thesis, Nov. 1988. University of California, Berkeley.

Oki85. B. Oki, B. Liskov and R. Scheifler, ‘‘Reliable Object Storage to Support Atomic Actions’’, *Proc. 10th Symp. on Operating System*

Prin., Operating Systems Review 19, 5 (December 1985), 147-159.

Oppen83. D. C. Oppen and Y. K. Dalal, ‘‘The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment’’, *ACM Trans. on Office Information Systems 1*, 3 (July 1983), 230-253.

Ousterhout85. J. Ousterhout, H. D. Costa, D. Harrison, J. Kunze, M. Kupfer and J. Thompson, ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System’’, *Proc. 10th Symp. on Operating System Prin., Operating Systems Review 19*, 5 (December 1985), 15-24.

Ousterhout88. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, ‘‘The Sprite Network Operating System’’, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

Ousterhout89a. J. Ousterhout and F. Douglis, ‘‘Beating the I/O bottleneck: A Case for Log-Structured File Systems’’, *Operating Systems Review 23*, 1 (Jan. 1989), 11-28.

Ousterhout89b. J. Ousterhout, ‘‘Why Aren’t Operating Systems Getting Faster as Fast as Hardware?’’, WRL Technical Note TN-11, Oct. 1989.

Parker83. D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser and C. Kline, ‘‘Detection of Mutual Inconsistency in Distributed Systems’’, *IEEE Transactions on Software Engineering*, May 1983, 240-247.

Popek85. G. J. Popek and B. J. Walker, *The LOCUS Distributed System Architecture*, MIT Press, Boston, 1985.

TCP81. J. Postel, ‘‘Transmission Control Protocol’’, *RFC 793*, Sep. 1981.

IP81. J. Postel, ‘‘Internet Protocol’’, *RFC 791*, Sep. 1981.

Powell77. M. L. Powell, ‘‘The DEMOS File System’’, *Proc. of the 6th Symp. on Operating System Prin.*, Nov. 1977, 33-42.

Presotto85. D. L. Presotto and D. M. Ritchie, ‘‘Interprocess Communication in the Eighth Edition Unix System’’, *USENIX Association 1985 Summer Conference Proceedings*, June 1985, 309-316.

Rees86. J. Rees, P. H. Levine, N. Mishkin and P. J. Leach, ‘‘An Extensible I/O System’’, *USENIX Association 1986 Summer Conference Proceedings*, June 1986, 114-125.

Renesse89. R. Renesse, J. M. Staveren and A. Tanenbaum, ‘‘The Performance of the Amoeba Distriubted Operating System’’, *Software Practice and Experience 19*, 3 (Mar. 1989), 223-234.

Rifkin86. A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah and K. Yueh, ‘‘RFS Architectural Overview’’, *USENIX Association 1986 Summer Conference Proceedings*, 1986.

Ritchie74. D. M. Ritchie and K. Thompson, ‘‘The UNIX Time-Sharing System’’, *Communications of the ACM 17*, 7 (July 1974), 365-375..

Ritchie84. D. Ritchie, ‘‘A Stream Input-Output System’’, *The Bell System Technical Journal 63,* 8 Part 2 (Oct. 1984), 1897-1910.

Rodriguez86. R. Rodriguez, M. Koehler and R. Hyde, ‘‘The Generic File System’’, *USENIX Conference Proceedings*, June 1986, 260-269.

Rowe82. L. A. Rowe and K. P. Birman, ‘‘A Local Network Based on the UNIX Operating System’’, *IEEE Trans. on Software Engineering SE-8,* 2 (Mar. 1982), 137-146.

Sandberg85. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, ‘‘Design and Implementation of the Sun Network Filesystem’’, *USENIX Conference Proceedings*, June 1985, 119-130.

Satyanarayanan85.
M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector and M. West, ‘‘The ITC Distributed File System: Principles and Design’’, *Proc. 10th Symp. on Operating System Prin., Operating Systems Review 19,* 5 (December 1985), 35-50.

Satyanarayanan89.
M. Satyanarayanan, ‘‘Coda: A Highly Available File System’’, *Proc. Second Workshop on Workstation Operating Systems*, Sep. 1989, 114-117.

Schroeder84. M. D. Schroeder, A. D. Birrell and R. M. Needham, ‘‘Experience with Grapevine: The Growth of a Distributed System’’, *ACM Trans. on Computer Systems. 2,* 1 (Feb. 1984), 3-23.

Schroeder85. M. Schroeder, D. Gifford and R. Needham, ‘‘A Caching File System For a Programmer’s Workstation’’, *Proc. 10th Symp. on Operating System Prin., Operating Systems Review 19,* 5 (December 1985), 25-34.

Scott89. M. Scott, T. LeBlanc and B. Marsh, ‘‘A Multi-User, Multi-Language Open Operating System’’, *Proc. Second Workshop on Workstation Operating Systems*, Sep. 1989, 125-129.

Sheltzer86. A. B. Sheltzer, R. Lindell and G. J. Popek, ‘‘Name Service Locality and Cache Design in a Distributed Operating System’’, *Proceedings of the 6th ICDCS,* May 1986, 515-522.

NFS85. *Sun’s Network File System*, Sun Microsystems, 1985.

RPC85. *RPC Protocol Specification*, Sun Microsystems, 1985.

Swinehart79. D. Swinehart, G. McDaniel and D. Boggs, ‘‘WFS: A Simple Shared File System for a Distributed Environment’’, *Proc. of the 7th Symp. on Operating System Prin.*, 1979, 9-17.

Swinehart86. D. Swinehart, P. Zellweger, R. Beach and R. Hagmann, ‘‘A Structural View of the Cedar Programming Environment’’, *ACM Trans. Prog. Lang and Systems 8,* 4 (Oct. 1986), 419-490.

Terry85. D. B. Terry, ‘‘Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments’’, PhD Thesis, Mar. 1985.

University of California, Berkeley.

Terry88. D. Terry and D. Swinehart, ‘‘Managing Stored Voice in the Etherphone System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

Thompson87. J. Thompson, ‘‘Efficient Analysis of Caching Systems’’, PhD Thesis, 1987. University of California, Berkeley.

Tomlinson85. G. M. Tomlinson, D. Keeffe, I. C. Wand and A. J. Wellings, ‘‘The PULSE Distributed File System’’, *Software-Practice and Experience 15*, 11 (Nov. 1985), 1087-1101.

Walker83a. B. Walker, ‘‘The LOCUS Distributed Operating System’’, *Proceedings of the 9th Symp. on Operating System Prin., Operating Systems Review 17*, 5 (Nov. 1983), 49-70.

Walker83b. B. Walker, ‘‘Issues of Network Transparency and File Replication in the Distributed File System Component of LOCUS’’, PhD Thesis, 1983. University of California, Los Angeles.

Welch86a. B. B. Welch, ‘‘The Sprite Remote Procedure Call System’’, Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

Welch86b. B. B. Welch and J. K. Ousterhout, ‘‘Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem’’, *Proc. of the 6th ICDCS*, May 1986, 184-189.

Welch88. B. B. Welch and J. K. Ousterhout, ‘‘Pseudo-Devices: User-Level Extensions to the Sprite File System’’, *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

Wilkes80. M. Wilkes and R. Needham, ‘‘The Cambridge Model Distributed System’’, *Operating Systems Review 14*, 1 (Jan. 1980).

Zwaenepoel85. W. Zwaenepoel, ‘‘Protocols for Large Data Transfers Over Local Networks’’, *Proc. of the 9th Data Communication Symposium*, Sep. 1985, 22-32.

Table of Contents

List of Figures

List of Tables