Pseudo-File-Systems

Brent B. Welch and John K. Ousterhout

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

ABSTRACT

This paper describes a facility that transparently extends the Sprite distributed file system to include foreign file systems and arbitrary user services. A pseudo-file-system is a sub-tree of the distributed hierarchical name space that is implemented by a user-level server process. A pseudo-file-system fits naturally into the Sprite distributed system; the server runs on one host and access from other hosts is handled in the same way as access to regular Sprite file servers. The pseudo-file-system interface is general enough to be used for version control systems, and access to database servers, as well as access to other kinds of file systems. We currently use a pseudo-file-system server to provide access to NFS file servers from Sprite workstations.

Pseudo-File-Systems

Brent B. Welch
John K. Ousterhout

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

Abstract

This paper describes a facility that transparently extends the Sprite distributed file system to include foreign file systems and arbitrary user services. A *pseudo-file-system* is a sub-tree of the distributed hierarchical name space that is implemented by a user-level server process. A pseudo-file-system fits naturally into the Sprite distributed system; the server runs on one host and access from other hosts is handled in the same way as access to regular Sprite file servers. The pseudo-file-system interface is general enough to be used for version control systems, and access to database servers, as well as access to other kinds of file systems. We currently use a pseudo-file-system server to provide access to NFS file servers from Sprite workstations. †

1. Introduction

Sprite [Ousterhout88] is a network operating system that is centered around its shared file system. The underlying distribution of the system is hidden behind the file system, which transparently provides access to local or remote files to all the Sprite hosts in the network. We designed the file system to cleanly handle local and remote file access through an internal kernel interface much like the *vnode* [Kleiman86] or *gnode* [Rodriguez86] interfaces in the UNIX[1] and ULTRIX[2] kernels. This kind of structure supports modular additions to the kernel to support other types of file systems. For

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, and in part by the National Science Foundation under grant ECS-8351961.

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

[2] ULTRIX is a registered trademark of Digital Equipment Corporation.

example, we could have provided access to NFS[3] [Sandberg85] file servers by adding an NFS file system type to the kernel. However, we decided instead to add a file system type that allows further extensions to the system to be implemented in user-level server processes instead of inside the kernel. We call the new file system type a *pseudo-file-system*.

Our main motivation for implementing pseudo-file systems was to provide access to existing NFS servers so that users could gradually switch over to using Sprite instead of UNIX. However, we think that pseudo-file-systems will also be useful for a variety of other applications where generality and ease of implementation are more important than achieving the absolute maximum performance. For example, a version control system might be implemented as a pseudo-file-system that automatically checks files in and out whenever they are used. Or, an archive service might represent itself as a pseudo-file system with a directory structure that indicates date of archival. In this case the performance overhead of the user-level implementation would be overshadowed by the cost of archive retrieval. Pseudo-file-systems provide a general mechanism for extending the naming and I/O structure of the file system with user-implemented applications.

The advantages of user-level implementation of system services have been promoted before by designers of message-based kernels [Cheriton84]. Debugging is easier because the server is an ordinary application and the standard debugging tools apply to it. The kernel remains smaller and more reliable. It is easier to experiment with new types of services. The pseudo-file-system approach has all of these advantages, plus it provides more structure than a message-based kernel. The file system orientation of the system means that there is a standard interface to the various system services so the environment is easy for users to understand. An archive service or a database, for example, can be accessed like the rest of the file system.

The file system support provided by the kernel allows a pseudo-file-system server to be simpler than a corresponding server in a pure-message based system. The distributed name space is managed by the operating system. The server implements its part of the name space and lets the system handle the problems of server location and remote access. The kernel does crash detection and supports automatic recovery of our file servers. The kernel buffers file data to optimize I/O. We are extending our recovery and caching mechanisms to support pseudo-file-system servers. Thus, Sprite is a "file-system-based" kernel that provides a standard interface to users and applications and provides more system support for user-implemented services than a message-based kernel.

A potential disadvantage of our approach, however, is that the performance of the pseudo-file-system will be degraded by its user-level implementation. Our measurements suggest that the performance degradation is as much as 50 percent for I/O intensive applications.

The remainder of this paper is organized as follows. Section 2 describes the way the Sprite distributed file system is organized. Section 3 describes the kernel structure that supports pseudo-file-systems. Section 4 describes our NFS pseudo-file-system and gives some performance results. Section 5 outlines our current work to extend the

[3] NFS is a registered trademark of Sun Microsystems.

kernel's caching and recovery systems to pseudo-file-systems. Section 6 reviews related work, and Section 7 gives our conclusions.

2. The Structure of the Distributed Name Space

Pseudo-file-systems are a natural extension of mechanisms already present in Sprite to support distribution. The file system is organized into *domains* controlled by different *servers*. Hosts that access the file system are called *clients*. A domain can be implemented by the local operating system kernel, it can be implemented at a remote host, or it can be implemented as a pseudo-file-system by a user-level process. Each domain is a sub-tree of the hierarchical name space, and the sub-trees can be nested arbitrarily to form the global hierarchy. The division of the name space into different domains is transparent to users and application programs. There is just one name space shared by all the Sprite hosts, and its distribution among servers is hidden by the operating system.

The distribution of the name space is managed by the operating system with a *prefix table* mechanism [Welch86a]. Each domain is identified by a prefix that is the name of the domain's top-level directory. The kernel on each host maintains a prefix table that is used to map a pathname to a domain, its server, and its type. The prefix tables are

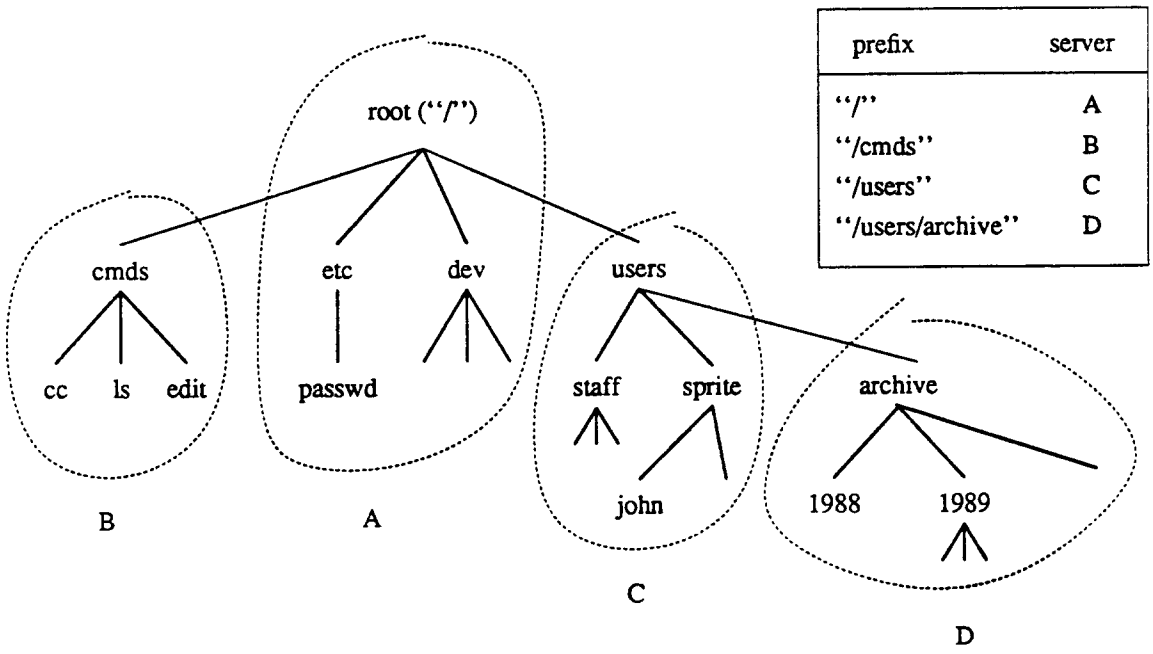

Figure 1. This shows the file system hierarchy and a prefix table that partitions the hierarchy into four domains. The distribution is transparent to applications. A domain's server might be the local operating system kernel, a remote Sprite kernel, or a user-level pseudo-file-system server. The server's type and a token that identifies the domain are also kept in the prefix table. For example, "/users/archive" can be implemented as a pseudo-file-system that presents a name space organized by date of archival.

managed as caches that contain information about the domains exported by a host and the domains currently in use by a host. The system automatically adds prefixes as new areas of the name space are accessed, and it automatically locates the server of a domain. Figure 1 shows an example of a file system divided into four domains and a prefix table that defines the division.

The use of the prefix tables is simple. During name lookup, absolute pathnames (those beginning at the root of the hierarchy) are compared against a client's prefix table and the longest matching prefix determines the domain. Operations on relative pathnames bypass the prefix table and are sent directly to the server of the process's current working directory. In both cases the server is passed a relative pathname and a token that identifies the pathname's starting point. The token comes from the prefix table entry, or from the open file information associated with the current working directory.

The layout of the domains is determined by *remote links* contained in the name space. When a server encounters a remote link during name lookup it returns a prefix and the remaining pathname to the client kernel. If the prefix is new to the client kernel then its prefix table is updated and the domain's server is located using a broadcast protocol. The lookup algorithm goes back and forth between the client kernel and various servers until the lookup completes. There is no centralized agent that has to know about the complete structure of the name space.

The prefix table mechanism was designed to support a distributed set of file servers, but it generalizes easily to support pseudo-file-systems. A pseudo-file-system is treated like any other domain. The pseudo-file-system server registers itself with the local kernel and the prefix table mechanism automatically incorporates the pseudo-file-system into the distributed name space. The benefit of this is that there is no visible distinction between a pseudo-file-system and other parts of the file system. Objects in a pseudo-file-system are named and accessed like the files and devices implemented by regular Sprite file servers.

3. Kernel Architecture

3.1. The File System Switch

Within the Sprite kernel, the file system is structured to handle different kinds of file systems by using an operation switch similar to the vnode or gnode switches in the UNIX and ULTRIX kernels. The prefix table is used by generic top-level procedures to determine the server for a pathname and its type: a local file system, a remote file system, or a pseudo-file-system. The file system type is used to branch through the switch to the proper naming procedure.

The remote file system type is used to access either a remote Sprite file server or a remote pseudo-file-system server. The kernel uses a network RPC protocol [Welch86b] to forward the request to the remote host. When a kernel receives a network request the token that identifies the prefix also indicates if the domain is a local file system or a pseudo-file-system. The naming operation switch is used again to branch to the correct routine. This is depicted in Figure 2.

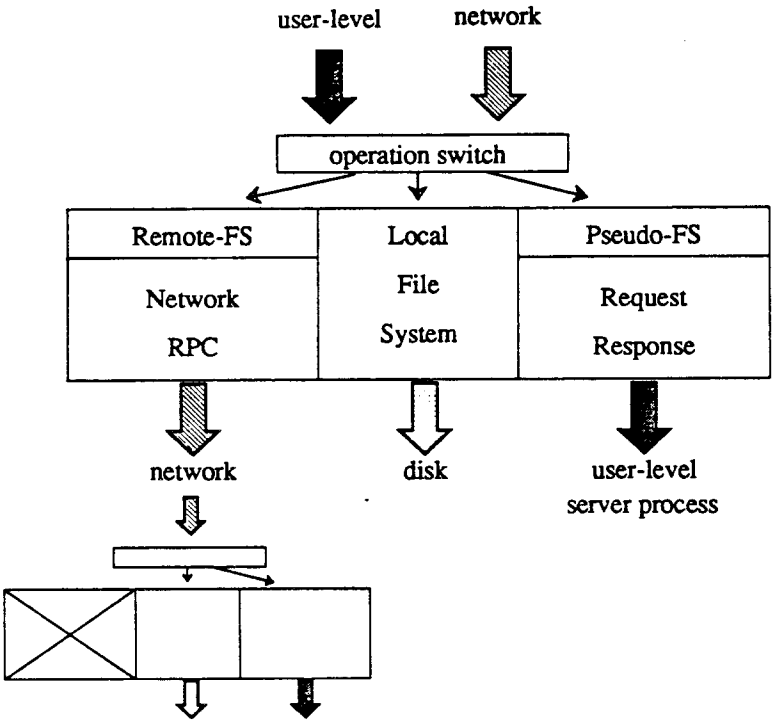

Figure 2. There are three types of file systems implemented in Sprite: local, remote, and user-level. There is a standard interface between the generic top-level file system procedures and the lower-level type-specific procedures so that differences among the types are hidden above that interface. The arrows entering at the top represent operations made from user-level via the system call interface, or from other hosts via network RPC. The arrows leaving the boxes represent operations that are forwarded to other Sprite hosts via network RPC, operations on the local disk, or operations forwarded to a user-level pseudo-file-system server. The miniature version of the picture connected to the network output arrow represents the use of the operation switch at a remote node to select either a local file system or a pseudo-file-system.

3.2. The Kernel-to-Server Interface

The kernel is in charge of forwarding operations on the pseudo-file-system up to the user-level server process. The operations can either originate from system calls made by user processes executing on the same host, or from network RPC requests that result from operations on the pseudo-file-system made by user processes at other hosts. The communication between the kernel and the server is implemented as a request-response protocol. The kernel formats a request message containing the parameters of the operation and passes this to the pseudo-file-system server. The server then implements the operation and responds with results and an error status.

A pseudo-file server typically has access to many request-response streams at any given time. For each domain managed by the server there is a single request-response stream used for all naming operations on the domain (see Table 1 for a listing of the naming operations). In addition, a separate request-response stream is established each

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 1. This lists the naming operations that are implemented by pseudo-file-system servers, and the DomainInfo operation that returns information about the whole pseudo-file-system.

time an object in the pseudo-file-system is opened; this request-response stream is used by the kernel to forward I/O operations to the server (see Table 2 for a list of the I/O operations). Each request-response stream appears to the server as a standard UNIX-like I/O channel. A pseudo-file server may multiplex itself among the various streams either as a single process that uses **select** to wait for incoming requests on all of the streams, or as a team of processes where each process services one stream.

The request-response mechanism used for pseudo-file-systems is a simple extension of the mechanism already in place to implement *pseudo-devices*. A pseudo-device is an object that appears like a file, but whose I/O operations are implemented by a user-level server process. The request-response protocol for pseudo-file-systems is identical to that for pseudo devices except that the naming operations in Table 1 do not exist for pseudo-devices. See [Welch88] for details of the request-response protocol.

Pseudo-Device Operations	
Read	Transfer data from an object.
Write	Transfer data to an object.
WriteAsync	Write without waiting for completion.
Ioctl	Make a special operation on an object.
GetAttr	Get attributes of an object.
SetAttr	Set attributes of an object.
Close	Close an I/O connection to an object.

Table 2. I/O operations on an object opened in a pseudo-file-system. The object is treated by the kernel like a pseudo-device. I/O operations on the object are forwarded to the pseudo-file-system server using the pseudo-device protocol.

3.3. A Flexible Name-to-Object Mapping

The file system architecture also makes it natural for a name in a pseudo-file-system to map to a regular file, a device, a pseudo-device, or a pipe. A source control system, for example, can map names with version numbers back to regular Sprite files. A rendez-vous service can map names to pipes in order to hook up processes. Thus the pseudo-file-system mechanism can be used to present a different name space for objects whose I/O functions are implemented by the operating system.

This flexible mapping of names to objects was designed to support remote device access. We had to be able to name a local device through a remote file server. We designed our architecture to clearly separate naming operations and I/O operations so this would be possible. A second operation switch is used for I/O operations, and the type used to branch through the switch is an object type like device, remote device, file, remote file, pseudo-device, remote pseudo-device, or pipe.

Mapping a name in the pseudo-file-system to an arbitrary object is implemented by passing open file descriptors between processes. In response to an open request, a pseudo-file-system server can open some existing object, i.e. a file, and then pass off its descriptor for the open file. This is done as an alternative to creating a request-response connection for the I/O operations as described in the previous sub-section. The kernel handles the case where the pseudo-file-system server and the process that generates the open request are on different hosts by using existing file system mechanisms that support process migration [Douglis87].

4. The NFS Pseudo-File-System

Our first application of pseudo-file-systems is a server that provides access to remote NFS file servers. The pseudo-file-system server translates file system operations into the NFS protocol and uses the UDP datagram protocol to forward the operations to NFS file servers. The pseudo-file-system server is very simple. There is no caching, of either file data or file attributes. The server process is single-threaded, and it multiplexes itself among requests for different files using the **select** system call. This avoids the cost of process creation when NFS files are opened, and eliminates the need to synchronize threads.

Figure 3 illustrates the communication structure for NFS access under Sprite. An interesting aspect of the NFS implementation is that the UDP network protocol, which is used for communication between the pseudo-file server and the NFS server, is not implemented in the Sprite kernel. Instead it is implemented by a user-level protocol server using the pseudo-device mechanism mentioned in Section 3. This approach adds additional overhead to NFS accesses, but illustrates how user-level services may be layered transparently.

Figure 3 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. There the regular request-response protocol is used to pass the operation along to the pseudo-file-system server.

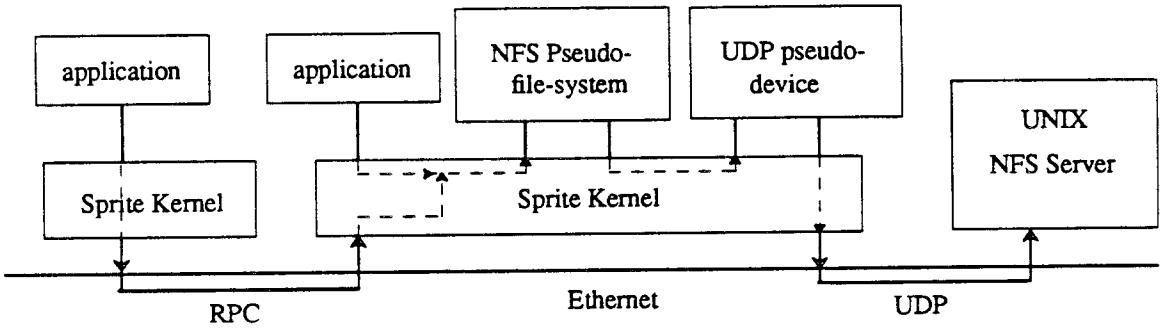

Figure 3. Two user-level servers are used to access a remote NFS file server. The first is the NFS pseudo-file-system server. In turn, it uses the UDP pseudo-device server to exchange UDP packets with the NFS file server. The figure also depicts requests to the NFS pseudo-file-system server arriving over the network from remote Sprite clients using the Sprite network RPC protocol. The arrows indicate the direction of information flow during a request.

4.1. NFS Performance

We measured the performance of our NFS pseudo-file-system with micro benchmarks that measured 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. This is especially true for our NFS access which uses two user-level servers for communication. However, when whole applications are run the effect of pseudo-file-system access is less pronounced. We view the current performance as an acceptable trade-off against the ease of implementing a pseudo-file-system with a user-level application.

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 operation system is SunOS 3.2 on the native 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 our distributed caching scheme [Nelson88].
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.

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 3. I/O performance when reading and writing a remote file. The file is in the server's main-memory cache when reading. Sprite uses 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.

The raw I/O performance for Sprite files, NFS files, and NFS files accessed from Sprite is given in Table 3. In all cases the file is in the file server's main memory cache. Ordinarily Sprite caches native Sprite files in the client's main memory. For the read benchmark we flushed the client cache before the test. For the write benchmark we 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.

We measured system-level performance of the NFS pseudo-file-system using the Andrew file system benchmark. This has been 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 [4] into an executable program. The results of running this benchmark are given in Table 4. We think a 33-41% slowdown relative

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

Table 4. 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.

[4] 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 [Nelson88].

to the native UNIX implementation is an acceptable trade-off against the cost of a kernel-level NFS implementation.

The user-level implementation of the UDP protocol has a large effect on the Sprite-NFS bandwidths given in Table 4. The cost to send data via a UDP packet and receive a one-byte acknowledgment packet is plotted in Figure 4. At small transfer sizes the overhead is over twice that of the UNIX kernel implementation. Larger transfers take about 25% longer.

The cost of sending data to a local pseudo-file-system (or pseudo-device) server is plotted in Figure 4 as the line labeled "local pdev". Note that the per-byte cost that comes from copying the data to the server is dominated by the base cost, which is about 3 msec. This is the time for two process switches and associated scheduling and synchronization overhead. This part of the kernel has remained untuned since its initial implementation and can mostly likely be improved.

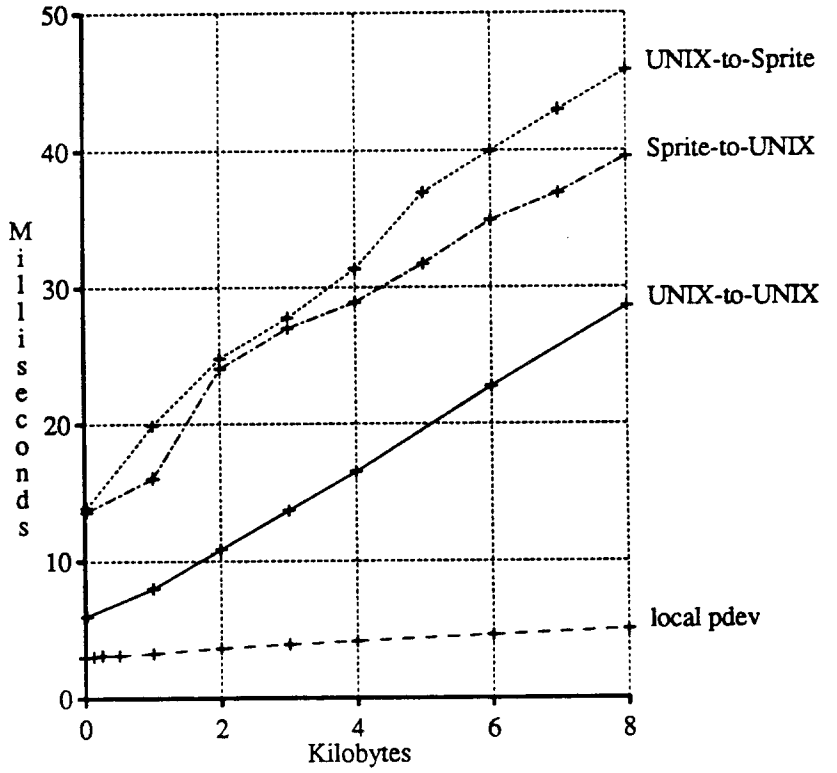

Figure 4. Timing of the UDP protocol. The receiver is always a UNIX process to model the use of UDP to communicate with the UNIX NFS server. Each Sprite-to-UNIX packet exchange requires two request-response transactions with the Sprite UDP server. The cost of accessing the UDP service via the pseudo-device 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.

5. Work In Progress

There are two additional aspects of pseudo-file-systems that are currently under development: file caching and automatic recovery. Sprite uses large file caches on both client and server machines, resulting in efficient file access even for diskless workstations [Nelson88]. The pseudo-file-system mechanism currently bypasses the caches, but we plan to modify the kernel so that blocks from pseudo-file-systems may be cached in the same way as blocks from "native" Sprite files. The pseudo-file-system server will define the caching policy, while the kernel will access the cache in response to I/O requests and do LRU replacement. This requires additional operations between the kernel and the pseudo-file-server for cache flushing and cache invalidation.

We are extending the kernel's recovery system for regular Sprite file servers to include pseudo-file-system servers. The kernel includes facilities for automatic detection of host crashes, recreation of the state of our file servers, and retry of operations with recovered servers. The system is based on state duplicated on the file servers and on other Sprite hosts. After a server crashes its state can be recovered from the other hosts. We are extending this facility to support recovery of pseudo-file-system servers by allowing them to register per-file state with their local kernel. The state gets propagated back to other hosts that have files open in the pseudo-file-system. This will allow us to recover either from a crashed server process or from the crash of the host running the server process.

6. Related Work

We classify pseudo-file-systems as a mechanism for system extension; a pseudo-file-system is a general mechanism that allows a new system service to be added to the system without modifying the operating system kernel. Many systems are only extensible by adding new code to the operating system kernel. This is true for many versions of UNIX, i.e. with the gnode and vnode architectures, and with the Version 8 streams facility [Ritchie84]. Other systems use the run-time library for system extensions [Rees86][Brownbridge82], or they use a message-based architecture and implement all services outside the kernel [Cheriton84].

The differences between pseudo-file-systems and these other approaches stem from features in the Sprite kernel that simplify the pseudo-file-system server process. The features implemented in the generic top-level layers of the file system do not have to be duplicated by the server. This includes the prefix table mechanism for distributed naming, blocking and non-blocking I/O, and (eventually) crash detection, automatic recovery, and data caching. Library-based systems and message-passing kernels, on the other hand, require the service, or library, to implement these functions.

7. Conclusion

Pseudo-file-systems are a natural extension of mechanisms already present in Sprite to support its distributed file system. The file system name space is structured into domains controlled by different servers. 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 in the kernel with the same mechanisms used to

access remote Sprite servers. The kernel also provides parameter checking, blocking and non-blocking I/O, caching and automated error recovery. (These last two features are currently being extended for use by pseudo-file-systems.) Thus the operating system provides the basic structure for a file system and a pseudo-file-system server can extend the structure.

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 NFS pseudo-file-system to be acceptable. We also anticipate further improvements by tuning our basic process switching and scheduling mechanisms.

References

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.

Cheriton84. D. R. Cheriton, "The V Kernel: A software base for distributed systems.", *IEEE Software 1*, 2 (Apr. 1984), 19-42.

Douglis87. F. Douglis, "Process Migration in Sprite", Technical Report UCB/Computer Science Dpt. 87/343, University of California, Berkeley, Feb. 1987.

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*, Feb. 1988, 51-81.

Kleiman86. S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *USENIX Conference Proceedings*, June 1986, 238-247.

Nelson88. M. Nelson, B. Welch and J. Ousterhout, "Caching in the Sprite Network File System", *Trans. Computer Systems 6*, 1 (Feb. 1988), 134-154.

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.

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.

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.

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.

Welch86a. 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.

Welch86b. B. B. Welch, "The Sprite Remote Procedure Call System", Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

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.