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 remote access to Sprite file servers. The pseudo-file-system interface is general enough to be used for version control systems, access to archival storage, 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. Performance results of the NFS server are given in order to evaluate the cost associated with user-level implementation of services. †

October, 1989

† 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. 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 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

---

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

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

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

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.

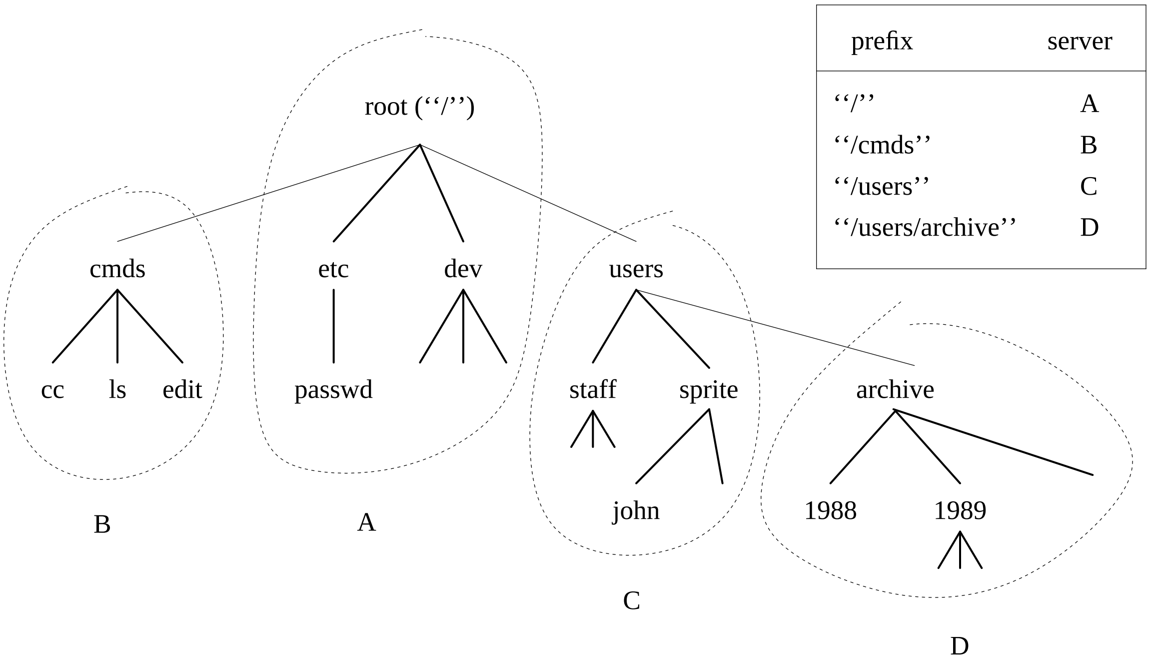
A disadvantage of our approach, however, is that the performance of the pseudo-file-system will be degraded by its user-level implementation. Our measurements suggest that the performance degradation is as much as 50 percent for I/O intensive applications. This indicates that systems that implement regular file service out of the kernel may suffer in performance. There is unavoidable overhead associated with context switching and message passing that is not incurred with kernel-resident services. Sprite optimizes the common case, regular file system activity, by implementing it inside the kernel, yet is also provides for transparent extension of the system via pseudo-file-systems.

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 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. 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 Sprite kernel 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 table records what domains are serviced by a host, and they also cache information about other



prefix	server
''/''	A
''/cmds''	B
''/users''	C
''/users/archive''	D

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.

domains being accessed by the host. The system automatically adds prefixes as new domains are accessed, and it automatically locates the server of a domain. Servers are located using broadcast, and there is no centralized agent that has to know the complete structure of the system. 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 directory at which to begin interpretation of the pathname. The token specifies a type, a server, and an identifier that is interpreted by the server. Tokens are obtained from the server when a prefix table entry is initialized, and also when the current directory of a process is defined.

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 cannot complete the lookup operation. Instead, it returns a prefix, which is stored in the remote link, 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 its prefix and a corresponding token 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. Naming and I/O Interfaces

The Sprite kernel architecture is built around two general interfaces, one for naming operations, which are listed in Table 1, and one for the I/O operations, which are listed in

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.

Table 2. Pseudo-file-systems were added by providing implementations of these interfaces that forward the operations out to the user-level server process. The distinction between the naming and I/O interfaces means that it is possible for the user-level server to implement either the naming interface or the I/O interface, or both. A server that only implements the I/O interface is called a *pseudo-device* server. In this case, the name for the pseudo-device is implemented by a Sprite file server, just as names for kernel-resident device drivers are implemented as special files. Sprite implements its X11 display server and its TCP/IP protocol server as pseudo-devices [Welch88]. The NFS server, which is described below, implements both interfaces according to the NFS protocol. An archive server or a version server can implement the naming interface, but let the Sprite kernel implement I/O operations once the correct file has been located.

There are three cases for naming operations. The server can be the local kernel, it can be remote, or it can be a user-level server. This is illustrated in Figure 2. The prefix table mechanism described in the previous section is used to determine the server and its type. Note that the case of a remote pseudo-file-system falls out naturally. It appears just like a remote Sprite file server to the client. After the Sprite kernel-to-kernel RPC

Pseudo-Device Operations	
Read	Transfer data from an object.
Write	Transfer data to an object.
WriteAsync	Write without waiting for completion.
Ioctl	Invoke 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. Read and write can be asynchronous in order to reduce context switching costs. Asynchronous reads are done via a read buffer, in which case the server never sees an explicit read request.

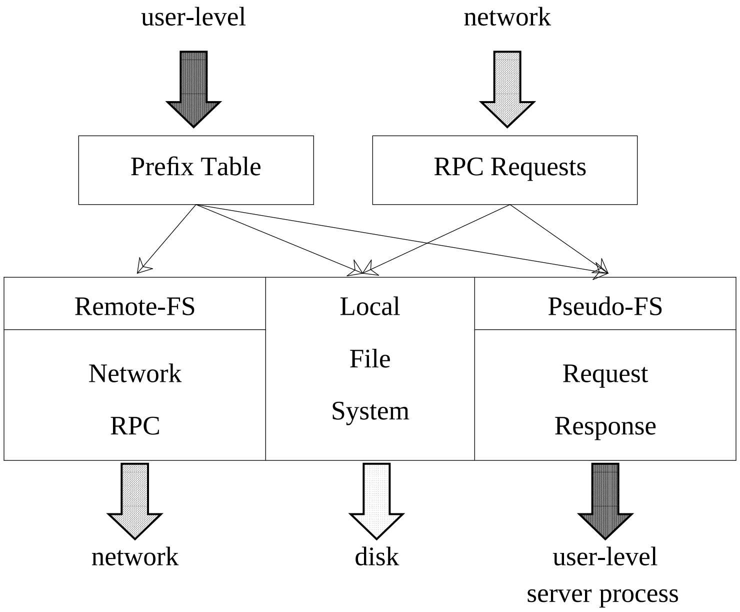


Figure 2. There are three types of naming domains implemented in Sprite: local, remote, and user-level. The arrows entering at the top represent operations made from user-level via the system call interface, and from other hosts via network RPC. The arrows leaving the boxes represent operations that 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.

protocol[Welch86b] forwards the operation to the remote host, the operation is forwarded up to the server process.

There are a number of cases for I/O operations that correspond to the different types of objects accessed via the file system interface. Figure 3 shows the various cases. Again, note the general way in which remote access is handled. A common set of RPC stub procedures are used to forward operations in the remote case (except for remote file operations that check the cache), and the operations go through the I/O interface at the remote site in order to branch to the correct case.

A pseudo-file-system server can avoid implementing the I/O interface by passing off open file descriptors in response to open requests. A version control system, for example, can generate a requested file version from the version history, open the file, and

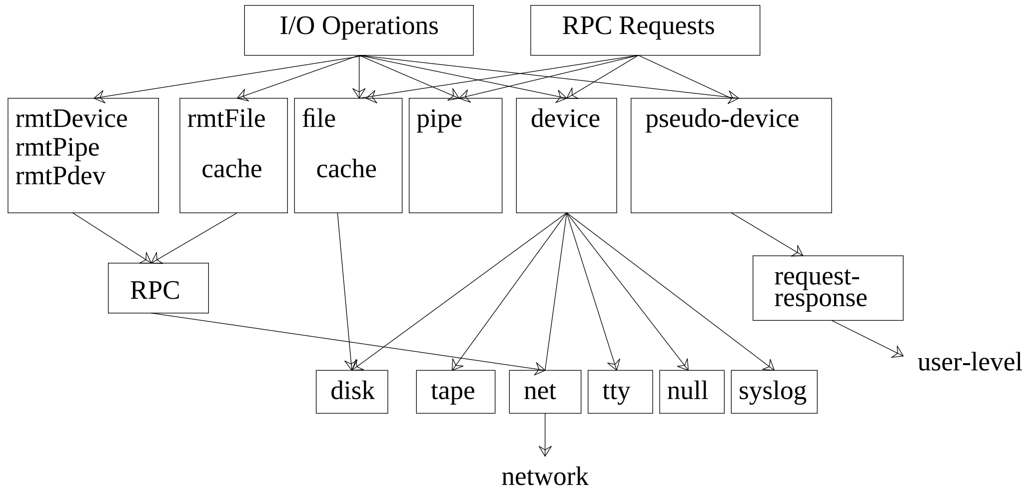


Figure 3. The kernel architecture for I/O operations. There are several implementations of the I/O interface corresponding to the different types of objects being accessed. A common set of RPC stubs handles the remote case, except for the remote file procedures that check in the data cache.

return its open file descriptor to the other process. Named pipes can be implemented by a pseudo-file-system server that simply creates regular UNIX pipes and passes off both ends in response to open requests. The ability to do this in the remote case is supported by existing file system mechanisms that support the migration of processes (and their open file descriptors) between hosts [Douglis87]. While these kinds of pseudo-file-systems are supported by the kernel, we haven't yet implemented any real applications that use them.

3.2. Request-Response Protocol

The communication between the kernel and a user-level 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. While this is much like RPC, the protocol is different than the network RPC protocol used between Sprite kernels. The network protocol is concerned with reliable delivery of messages on an unreliable network, and with efficient use of network packets. The kernel to user-level request-response protocol is optimized to reduce context switching and to eliminate the use of kernel-resident buffers.

The server process allocates a *request buffer* and (optionally) a *read buffer* for each *request-response stream*. These buffers are in the server process's address space, and their size is chosen by the server. The kernel copies request messages directly into the request buffer, at which point a **read** by the server returns a <*firstByte, lastByte*> pair of values that define where the message is located in the request buffer. The server responds with an **ioctl** that indicates the return status of the operation and the size and location of any return data. The kernel copies the return data directly to the client process's address space, and the use of kernel buffering is eliminated.

Context switching is reduced by using asynchronous reads and writes. A server can enable asynchronous writes, in which case the kernel doesn't wait for a reply after copying a write request and the associated data into the server's request buffer. The server has to accept all the data being written as there is no opportunity to return an error code. Many write requests can be buffered before switching to the server process. A read buffer can be used to reduce context switching during reads. The server fills the read buffer as data is generated, and the kernel empties the buffer to satisfy read requests. If the read buffer is used then the server doesn't see explicit read requests, and <*firstByte, lastByte*> pointers are used to synchronize over the read buffer.

A pseudo-file-system 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. In addition, a separate request-response stream can be established each 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. Each request-response stream appears to the server as a standard UNIX-like I/O channel, and each stream has its own request buffer. The server uses **read** to learn the current <*firstByte, lastByte*> values for the read buffer and the request buffer, and it uses **ioctl** to reply to requests and update the pointers. A pseudo-file-system server may multiplex itself among the various streams either as a single process that uses **select**, or as a team of processes where each process services one stream. The request-response protocol and its performance are examined in more detail in [Welch88].

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, so all operations are forwarded to the NFS server. 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 4 illustrates the communication structure for NFS access under Sprite. An interesting aspect of the NFS implementation is that the UDP network protocol, which is

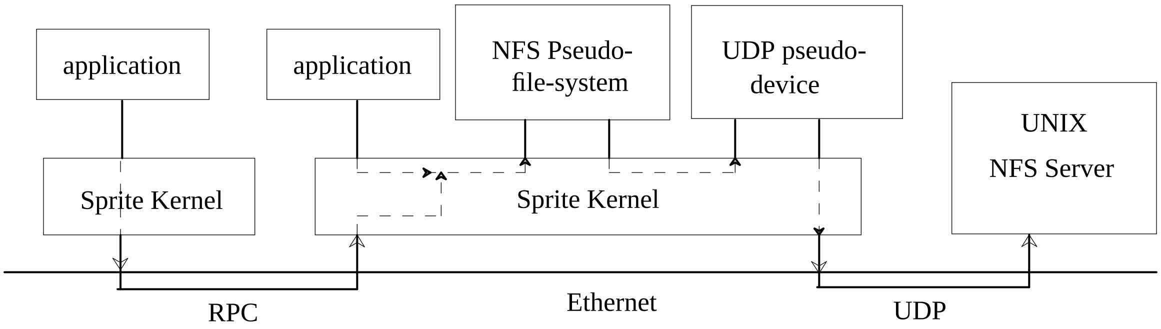


Figure 4. Two user-level servers are used to access a remote NFS file server. The first is the NFS pseudo-file-system server. In turn, it uses the UDP pseudo-device server to exchange UDP packets with the NFS file server. The figure also depicts requests to the NFS pseudo-file-system server arriving over the network from remote Sprite clients using the Sprite network RPC protocol. The arrows indicate the direction of information flow during a request.

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 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 4 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. Thus the kernel-to-kernel and kernel-to-user protocols can be composed together to handle the case of a remote-user-level server.

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 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 Sprite's distributed write-back caching system [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 — Like the previous case, except that the pseudo-file-system server is on a different host than the application so the kernel-to-kernel RPC protocol is also used.

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 files are cached in the client's main memory, too. 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 slightly 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 NFS pseudo-file-system server enables asynchronous writes in order to obtain the performance given in Table 3. It declares a request buffer large enough for two 8K writes plus the message headers. While the server is waiting for a UDP reply packet its client can issue a second write request. This buffering improved the raw NFS write bandwidth from about 9K/sec to 40K/sec.

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.

We measured system-level performance of the NFS pseudo-file-system using the Andrew file system benchmark, which 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 [4] into an executable program. The results of running this benchmark are given in Table 4. The 33-41% slowdown relative to the native UNIX implementation illustrates the performance tradeoff when implementing services at user-level. This performance hit is acceptable to us because most of our files are on much higher performance Sprite servers. NFS access lets us access the (fewer and fewer) files still kept on NFS servers in our network. Furthermore, we can run all the NFS pseudo-file-system servers on one Sprite host, whereas a kernel NFS implementation would inflate the size of every Sprite kernel. Thus we have high performance for most work, and transparent, although slower, access to foreign systems, if needed.

The user-level implementation of the UDP protocol has a large effect on the Sprite-NFS bandwidths given in Table 4. (Recall that Sprite uses its own kernel-to-kernel RPC protocol for inter-Sprite communication, so the performance of UDP is not ordinarily

[4] The version we used here has been modified to eliminate machine dependencies, the main one being the compiler used. This standardized Andrew benchmark uses the gcc compiler generating code for the SPUR, so these results are not directly comparable with those reported in [Howard88] and [Nelson88].

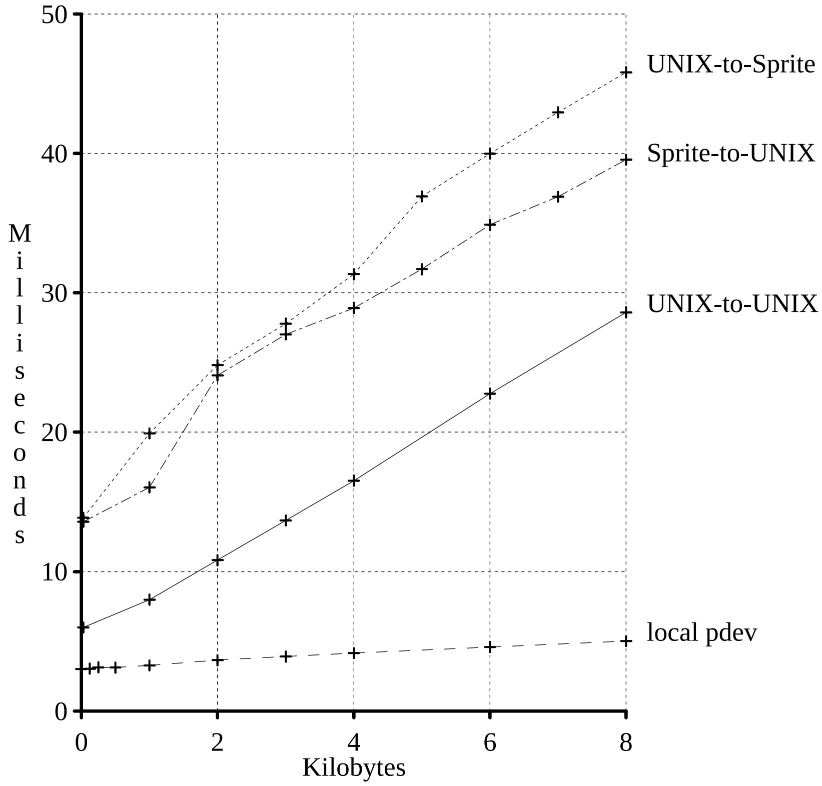


Figure 5. Timing of the UDP protocol. The receiver is always a UNIX process to model the use of UDP to communicate with the UNIX NFS server. Each Sprite-to-UNIX packet exchange requires two request-response exchanges with the Sprite UDP server. The cost of one request-response exchange is given by the line labeled ‘‘local pdev’’. Doubling this cost accounts for most of the difference in UDP performance. Receiving large packets on Sprite is slower because IP fragment reassembly cannot be done in the interrupt handler as it is in UNIX.

important.) The cost to send data via a UDP packet and receive a one-byte acknowledgment packet is plotted in Figure 5. At small transfer sizes the overhead is a little over twice that of the UNIX kernel implementation. Larger transfers take about 25% longer. The overhead is due mainly to the cost of the request-response protocol. However, the performance of receiving large packets is further reduced because IP packet reassembly is not done in the interrupt handler as it is in UNIX. Each IP fragment has to be passed up to the user-level server for reassembly.

The cost of one request-response is plotted in Figure 5 as the line labeled ‘‘local pdev’’. There is one request-response for each UDP packet sent, so there are two request-response exchanges in the UDP benchmark plotted in Figure 4. Note that the cost of request-response is dominated by the base cost, which is about 3 msec, and not the per-byte copy cost. The base cost is the time for two process switches, 4 system calls, a VM mapping operation, and scheduling and synchronization overhead. The Sprite scheduler has remained untuned since its initial implementation and can mostly likely be improved. There is also a relatively high VM mapping cost, (about 400 microseconds!), needed to implement the copy between two user processes on the Sun3 hardware. Only one process’s address space is accessible at one time, so the server’s request buffer is mapped into the kernel during the copy. This mapping is not cached, so its cost is incurred for each request-response.

5. Future Work

There are two additional aspects of pseudo-file-systems that have been designed but not implemented: data 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

† The author is frantically trying to get his dissertation written, of which pseudo-file-systems are only a small part.

requests and do LRU replacement. This requires additional operations between the kernel and the pseudo-file-system server for cache flushing and cache invalidation.

We plan to extend 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 recovery system is based on state duplicated on the file servers and on other Sprite hosts. After a crashed file server reboots, its state is recovered from the other hosts. We want to extend this facility to support recovery of pseudo-file-system servers by allowing them to register per-file state with their local kernel. For NFS, this is simply the NFS file ID of each file. The state has to be 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].

V is a message-based system that moves all higher-level services, like the file sysytem, outside the kernel. Like Sprite, V uses a prefix table mechanism to identify servers. The main difference between the Sprite and V prefix table mechanisms is the remote links in Sprite. These identify prefixes and replace the administrative manager required in V [Cheriton89]. V also has a standard I/O interface, UIO, which is comparable to the Sprite I/O interface [Cheriton87] The main difference between the I/O interfaces concerns the way I/O operations are blocked. Blocking I/O is implemented at a high level in Sprite in order to support **select** on objects located throughout the network. In this case it is not appropriate to block a process within the implementation of an object, i.e. on a remote host. All long term waiting is done on the client, and the kernel's waiting primatives handle races that are possible in a distributed environment.

The Watchdog facility proposed by Bershad provides similar functionality to Sprite pseudo-file-systems [Bershad88]. A watchdog process can guard a file by intercepting certain operations. The implementation is not as well integrated into the file system architecture as pseudo-file-system are; special case checks are required in each file system operation to see if they are ''guarded''. There is also no support for the remote case, which falls out naturally in the Sprite file system architecture.

Thus Sprite shares many ideas with other file systems. However, it represents a hybrid between the monolithic kernels of traditional operating systems, and the message-passing kernels of more recent systems. Sprite appears like a monolithic kernel from the outside, and it handles the important case of remote access with an efficient kernel-to-kernel RPC network protocol. However, the pseudo-file-system and pseudo-device mechanisms provide an ''escape hatch'' where additional functionality can be added to the system by user-level applications. Furthermore, 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, remote access, blocking and non-blocking I/O, and (eventually) crash detection, automatic recovery, and data caching.

7. Conclusion

There are two main conclusions to make regarding pseudo-file-systems. The first is that because the Sprite kernel is carefully structured to support a distributed system, it was very straight-forward to integrate a user-implemented service into the 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 to a pseudo-file-system is handled in the kernel with the same network RPC protocol 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.) The main addition to support pseudo-file-systems was a request-response protocol for passing operations up to the user-level server process.

The second conclusion is that there is a significant performance penalty for user-level implementation of services. The 30%-40% penalty for NFS access from Sprite can either be viewed pretty good or pretty awful, depending on your point of view. If you want transparent access and ease of implementation, then this performance hit isn't so bad. If you want high-performance, however, you should implement your service inside the kernel. This is the approach taken in Sprite, where access to remote Sprite file servers is highly optimized by a kernel implementation and write-back caching, and extra features, such as NFS access, can be added transparently at user-level, but at a reduced level of performance.

References

Bershad88. B. N. Bershad and C. B. Pinkerton, ‘‘Watchdogs - Extending the UNIX File System’’, *USENIX Association 1988 Winter Conference Proceedings*, Feb. 1988, 267-275.

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.

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.

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 6*, 1 (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.