Pseudo Devices: User-Level Extensions to the Sprite File System

Brent B. Welch
John K. Ousterhout

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

Abstract

A pseudo-device is a mechanism in the Sprite network file system that lets a user-level server process emulate a file or I/O device. Pseudo-devices are accessed like regular files or devices, and they exist in the file system name space. Pseudo-devices are implemented by transparently mapping client operations on the pseudo-device into a request-response exchange with a server process. The interface to pseudo-devices is general enough to be a transport mechanism for a user-level RPC system. It also provides a stream-oriented interface with write-behind and read-ahead for an asynchronous connection between clients and server. Sprite uses pseudo-devices to implement at user level its terminal drivers, the internet protocol suite, and the X-11 window system server. The pseudo-device implementation provides as fast or faster communication, both local and remote, than a UNIX UDP socket connection. †

April, 1988

1. Introduction

This paper describes pseudo-devices, a mechanism used to integrate user-level server processes into the Sprite operating system's distributed file system [Ousterhout88]. The distinguishing feature of pseudo-devices is that they appear in the file system name space and behave like regular files or devices. However, the operations on a pseudo-device are actually forwarded to a user-level process, called the server, which can implement them in any way it chooses. The motivation for pseudo-devices is to be able to move services traditionally found in the operating system kernel out to user-level processes. This keeps the size of the kernel down and it makes services easier to develop

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

and debug. Sprite uses pseudo-devices to implement several system services including terminal drivers, the X windows server [Scheifler86], and the TCP/IP protocol family [IP81].

There are two advantages to using pseudo-devices to access user-level services: communication is via a traditional interface, and naming is via the existing file system name space. Pseudo-devices are accessed with the same open(), read(), write(), iocontrol(), select(), and close() procedures used to access regular files and I/O devices. Read() and write() provide a byte stream interface which can be fully asynchronous with write-behind and read-ahead. Iocontrol() is synchronous and can be used as a transport mechanism for a Remote Procedure Call (RPC) system.

Using pseudo-devices, there is no need to invent a new name space to support user-level services. Names for pseudo-devices are protected, manipulated, and browsed in the same way as for other kinds of files. The Sprite file system provides name transparency across the network, so a pseudo-device can be accessed from any host in the network.

The remainder of the paper is organized as follows. Section 2 gives some background on the Sprite network operating system. Section 3 describes the pseudo-device interface from a client's point of view, and includes examples of services implemented as pseudo-devices. Section 4 describes the server's view of pseudo-devices and the implementation of the interface. Section 5 gives some performance measurements, including comparisons with other systems. Section 6 describes related work. Section 7 gives a summary and our conclusions.

2. Sprite Background

Sprite is a network operating system that we have developed from scratch at U.C. Berkeley over the last few years. Its design has been influenced by three things: multiprocessors, local area networks, and large physical memories. To support execution on a multiprocessor the Sprite kernel is multi-threaded internally, and Sprite supports shared memory between cooperating user processes. To support networks, Sprite uses a custom kernel-to-kernel RPC system [Welch86a] and a shared network file system that provides location transparent access [Welch86b] to files and I/O devices around the network. To exploit large physical memories, a distributed file caching scheme is used for high performance file access in an environment of diskless workstations [Nelson88]. The Sprite user interface is much like UNIX†, and for the purposes of this paper one can think of Sprite as a UNIX with a transparent network file system. Pseudo-devices could easily be implemented in a traditional UNIX kernel, and Section 6 describes some similar mechanisms that have been added to UNIX kernels.

3. Pseudo-device Clients

3.1. The Client Interface

Regular processes, called ''clients'', access a pseudo-device the same way they access regular files and I/O devices. Open() names the pseudo-device and sets up a

---

UNIX is a trademark of AT&T.

stream to it. Close() releases the stream. Write() and read() transfer data to and from the pseudo-device. Select() is used to wait until the pseudo-device is ready for I/O. Iocontrol()† is used for operations particular to the pseudo-device. These operations are identical, both in syntax and semantics, to the operations used for other files, so the implementation of the pseudo-device is transparent to the client.

3.2. Examples of Pseudo-Device Clients

Pseudo-devices are currently used for three purposes in Sprite: terminal drivers, network protocols, and window server communication. For example, for each terminal there is a pseudo-device and corresponding server process. Client processes make read() and write() requests on the pseudo-device instead of the terminal's serial line. The server implements the client's requests by manipulating the terminal's serial line in raw mode, and provides the full suite of 4.3 BSD ioctl() calls and line-editing functions such as backspace and word erase. The terminal driver is built as a library package so the same code is also used for managing rlogin streams and terminal-emulator windows. In this case, the pseudo-device mechanism provides a generalization of the 4.3BSD pseudo-tty facility.

The second use of pseudo-devices is to implement network protocols at user-level, TCP/IP in particular. While Sprite uses a custom network RPC protocol for kernel-to-kernel communication [Welch86a], the TCP/IP protocols are used to interface to non-Sprite systems, i.e. for mail service, remote logins, and remote file transfer. Client processes read and write a pseudo-device to use TCP, and the user-level server process implements the full protocol by reading and writing packets over the raw ethernet. In this case the internet pseudo-device provides the transport mechanism for a remote-procedure-call-like facility. A library package used by clients emulates the socket

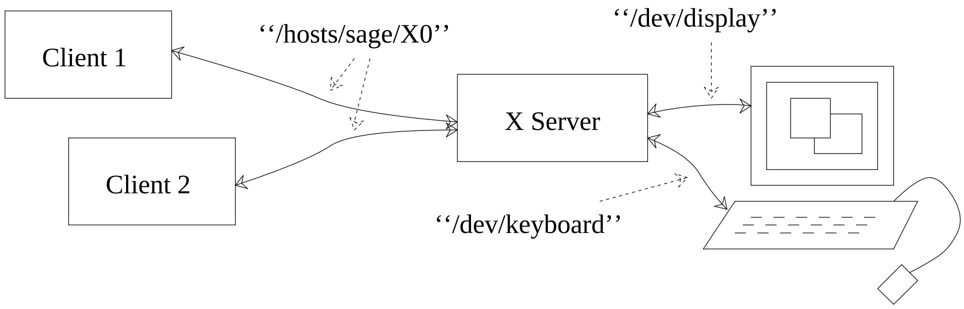

Figure 1. The pseudo-device ''/hosts/sage/X0'' is used by clients of the window system to access the X window server on workstation ''sage''. The server, in turn, has access to the display and keyboard.

---

† Iocontrol() is a super-set of the standard UNIX ioctl() system call. Ioctl() takes only a command ID and a buffer as arguments. Iocontrol() takes a command ID, and 4 arguments that specify the size, and location, of an input and output buffers.

operations by issuing iocontrols on the internet pseudo-device. The internet server defines iocontrol operations for socket calls like bind(), listen(), connect() and accept(). The iocontrol input buffer is used to pass arguments from the client to the server, where the socket procedure is executed. The output buffer is then used to return results back to the client. Each library procedure in the client is simply a stub that copies arguments and results into and out of buffers and invokes the iocontrol.

The third use of pseudo-devices is to implement the X window server at user-level. For each display in the network there is a pseudo-device and a server process. Access to remote displays is not a special case because the file system provides network transparency. The X server controls the display and multiplexes the mouse and keyboard among clients, as shown in Figure 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 4.2, provides an asynchronous interface between the window server and its clients to reduce context switching overhead.

4. Pseudo-device Server Implementation

The server for a pseudo-device is much like the server for 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. 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 subsections describe the implementation of this 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.

4.1. The Server's Interface

The server for a pseudo-device is established when it opens the pseudo-device with the PDEV_SERVER flag. This open returns a *control stream* to the server. The server listens on the control stream for messages issued by the kernel each time a client opens the pseudo-device. These messages identify a new *request stream* the server gets for use in communicating with the client. Thus the server process has one control stream used to wait for new clients, and one request stream for each open() by a client process. These are shown in Figure 2.

The server accepts or rejects the client's open attempt when it handles the first request (an open) on a request stream. (Details of handling requests are given in the next sub-section.) If a client process subsequently forks (creates a new process), the new process shares the stream to the pseudo-device, and the server is not contacted. This is much cheaper than creating a new request stream each time a client process forks, and it maintains the UNIX semantics of shared streams. To the server, however, this means that there may be more than one client process using a request stream.

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

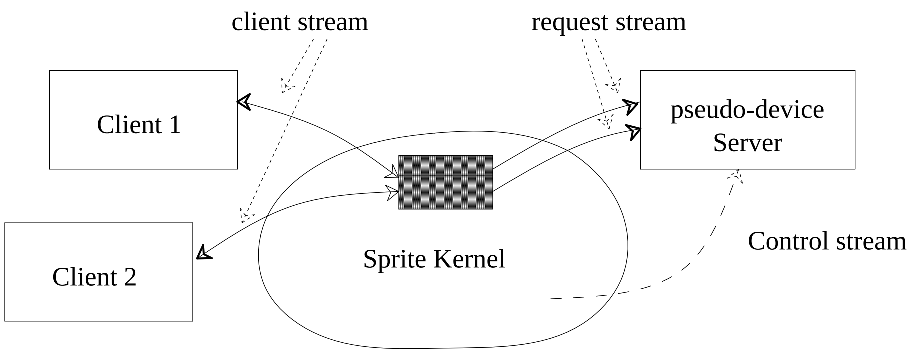

Figure 2. The control stream originates from the kernel and is used to pass new request streams to the server process. The client's streams are connected to corresponding request streams by a request-response protocol, which is represented by the black box in the figure.

reply message. Each request message includes the operation (open, close, read, write, iocontrol) and its associated parameters, which may include a block of data. Table 1 describes the contents of the request messages. The server replies to requests by making an iocontrol() call on the request stream. The iocontrol specifies the return code for the client's system call, and the size and location of any return data for the call (i.e. data being read). The kernel copies the reply data directly from the server's buffer to the client's.

Request Message Contents	
All Messages	Operation ID Process ID Input size Reply size
open	User ID (no data)
read	Byte offset (no data)
write	Byte offset Data block
iocontrol	Command ID Data block
close	(nothing)

Table 1. All request messages have a standard header that indicates the operation (open, read, write, iocontrol, close), the client's process ID, the the size of the input and result data. Additionally, each operation has its own specific parameters and, optionally, a variable size block of data.

The kernel passes request messages to the server using a *request buffer*, which is in the server process's address space, and an associated pair of pointers, *firstByte* and *lastByte*, that are kept in the kernel and indicate valid regions of the buffer. With this buffering scheme the server does not read the request messages from its request stream. Instead, the kernel puts request messages directly 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. After processing the message(s) found between firstByte and lastByte the server updates firstByte with an iocontrol. This buffering scheme permits the kernel to place several messages in the request buffer without waiting for each to be processed; the server can then process all of them at once, without requiring a context switch for each.

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 the 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 requests without the need for a context switch into and out of the server for each. On multiprocessors, write-behind permits concurrent execution between the client and server.

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 is shown in Figure 3. No single request may be larger than the server's buffer: oversize writes are split into multiple requests, and oversize iocontrols are rejected. If a write is split into several requests, the request stream is locked to preserve the atomicity of the original write.

Read performance can be optimized by using a *read-ahead buffer*. The server fills the read-ahead buffer, which is again in its own address space, and the kernel copies data out of it without having to switch out to the server process. Synchronization is done with firstByte and lastByte pointers as with the request buffer. In this case the server process updates lastByte after it adds data, and the kernel moves firstByte to reflect client reads.

To summarize the buffering scheme, the server has a request buffer associated with each request stream, and possibly a read-ahead buffer for each stream. These buffers are allocated by the server, and an iocontrol() call is used to tell the kernel the size and location of each buffer. The kernel puts request messages directly into the request buffer. 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 (the request firstByte and read-ahead lastByte) by making an iocontrol() on the request stream. The iocontrol calls available to the server are summarized in Table 2.

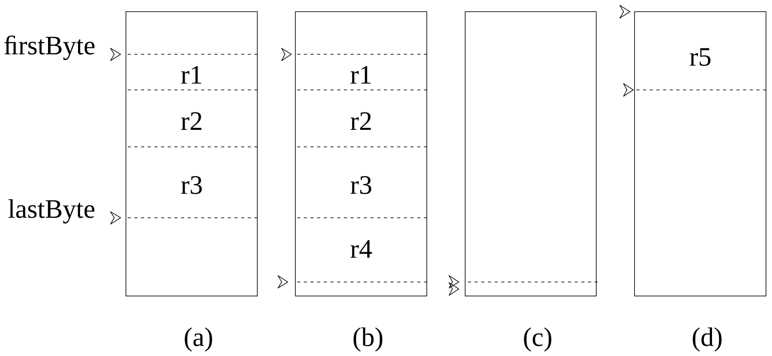

Figure 3. This figure shows the way the firstByte and lastByte pointers into the request buffer are used. Initially there are 3 outstanding requests in the buffer. The subsequent pictures show the addition of a new request, an empty buffer (the server has processed the requests), and finally the addition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

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 2. The server uses these iocontrol() 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.

4.3. Waiting for I/O

Normal I/O devices include a mechanism for blocking processes if the device isn't 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 device is ''ready''. One possibility would be for the kernel to make a request of the server whenever it needs to know whether a pseudo-device is ready, such as during read, write, and select† calls. We initially implemented pseudo-devices 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 of the pseudo-devices were not ready, so the servers were invoked needlessly.

† The select() call is used to wait for several I/O streams at once. Each stream is identified by a small integer, and select takes as an argument a bitmask that has a bit set corresponding to each stream that is being waited on. Select is used to wait for streams to become readable, writable, or to have exceptional conditions.

We subsequently re-implemented pseudo-devices so that the kernel maintains three bits of state information for each pseudo-device, corresponding to the readable, writable, and exception masks for the select kernel call. The pseudo-device server updates these bits with each reply iocontrol and can also change them with the IOC_PDEV_READY iocontrol. This mechanism allows the kernel to find out whether a pseudo-device is ready without contacting the server and 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.

4.4. Network Transparency

The Sprite file system provides a network-wide name space and remote device access so the pseudo-device server is not constrained to execute on the same host as its clients. An operation by a client on a remote pseudo-device is first shipped to the host running the pseudo-device server using the kernel's network RPC. The kernel RPC stub on the server's host then calls the regular pseudo-device routines to carry out the request-response exchange with the pseudo-device server. This is all transparent to both the client and the server processes.

4.5. Crash recovery

The client process is dependent on the server process to faithfully implement the pseudo-device. If the server hangs without returning a reply, then the client will also hang until the server process is killed or replies. If either the client or the server process dies, or their host crashes, then their stream to the pseudo-device automatically gets closed. When the client closes, the server gets a regular close request. If the server dies then current or future attempts by the client to use its stream will fail. Note that this applies even when write-behind is enabled.

5. Performance Review

This section of the paper presents a few performance measurements for pseudo-devices. There are a number of contributions to the communication cost: system call overhead, context switching, copy costs, network communication, synchronization, and other software overhead. For comparison, UNIX TCP and UDP sockets, and pipes under both Sprite and UNIX were also measured. 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 make a reply, and one to update the firstByte pointer into the request buffer. With the other mechanisms the client makes two system calls: one to prod the server and another to wait for a response. The server makes two system calls as well: one to respond to the client,

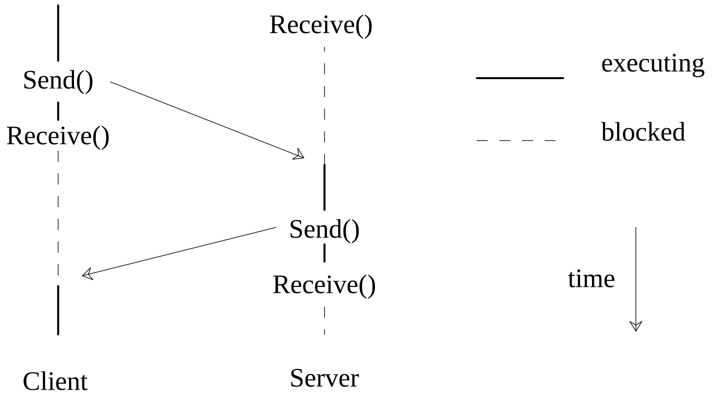

Figure 4. This shows the flow of control between two processes that exchange messages. Initially the server is waiting for a message from the client. The client sends the message and then blocks waiting for a reply. The client executes again after the server waits for the next request. Each benchmark has a similar structure, although different primitives are used for the Send() and Receive() operations shown here.

and one more to wait for the next request. The flow of control is shown in Figure 4.

Table 3 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. In the second half of the table the client

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 3. 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 iocontrol() call on a pseudo-device, UDP exchanges 1 byte using a UNIX UDP datagram socket, and TCP exchanges 100 bytes (to avoid buffering in the protocol) using a UNIX TCP stream socket.

and server processes are on different hosts so there are network costs.

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 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.

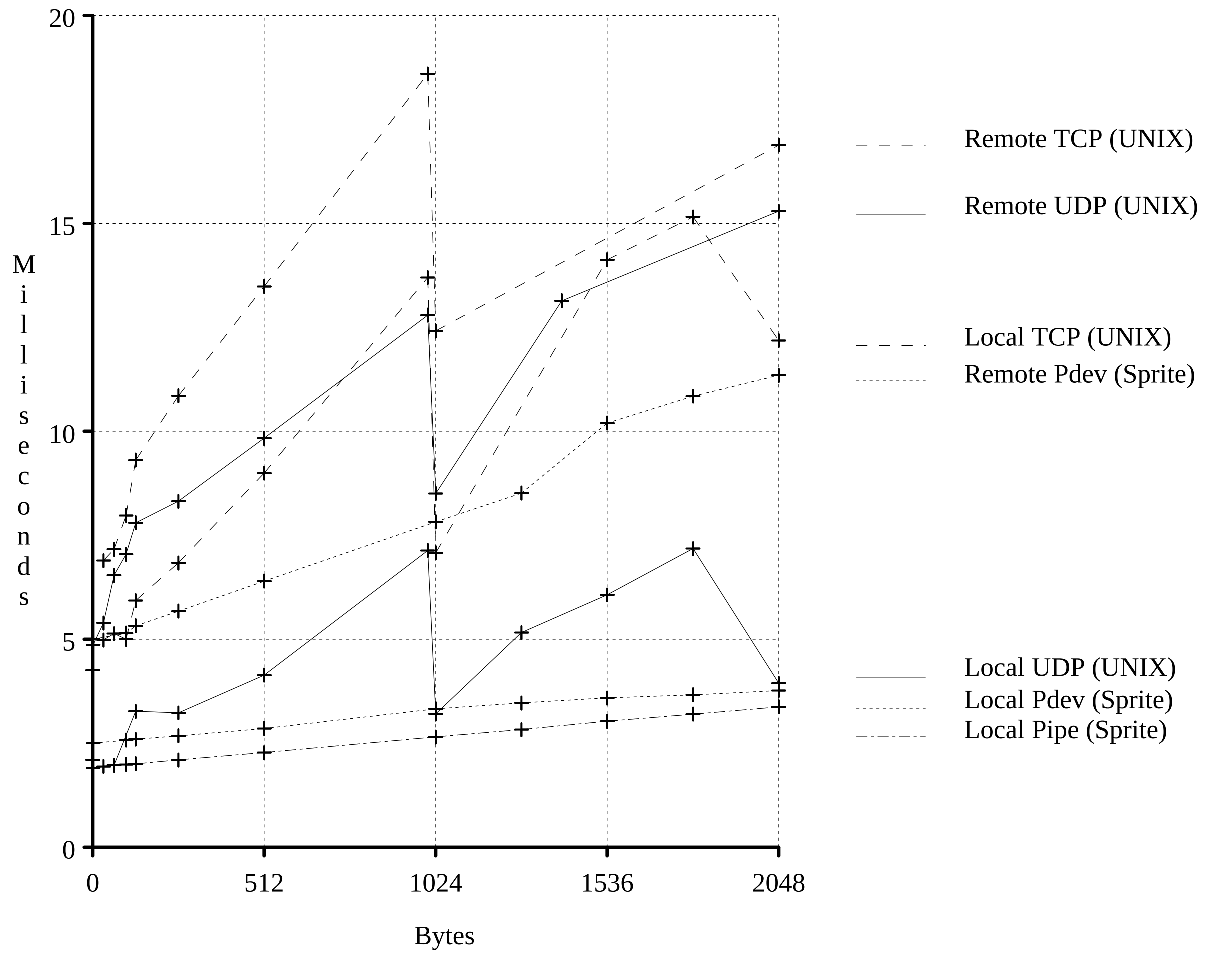

Figure 5. Elapsed time per exchange vs. bytes transferred, local and remote, with different communication mechanisms: Pseudo-devices, Sprite pipes (local only), and UNIX UDP and TCP sockets. The bytes were transferred in both directions. The shape of the UDP and TCP lines is due to the buffering scheme used in UNIX; chains of 112-byte buffers are used until a message is 1024 bytes, at which point chains of 1024-byte buffers are used.

Figure 5 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 buffering scheme used in UNIX; chains of 112 byte buffers are used until a message is 1024 bytes, at which point chains of 1024 byte buffers are used.

The Sprite mechanisms have nearly constant per-byte costs. The unrolled byte copy routine used by the kernel takes about 200 microsends per kbyte. Data is copied four times using pipes (there is an intermediate kernel buffer) and the measured cost is about 800 microsends per kbyte. Data is copied twice using pseudo-devices, and we expect a per-kbyte cost of 400 microseconds. This is obtained when transferring between 1-kbyte and 2-kbytes, but we are not sure of the reason for the slightly higher cost at smaller transfer sizes.

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. This adds about 2.2 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 4 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 doesn't return an explicit reply. The table also gives the optimal number of context switches possible, which is a function of the size of each request, and the actual number of context switches taken during 1000 iterations. 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 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 scheduler preempts the client too soon so there are twice as many context switches as expected.

6. Related Work

There are a number of features in existing operating systems that provide similar functionality to pseudo-devices, or that can be used to build up similar functionality. They fall into three categories, byte-stream mechanisms, message-based or RPC systems, and device-like mechanisms.

Message systems, including RPC implementations, are useful for implementing services outside the operating system kernel. However, they are usually not integrated into the file system name space, so an extra name service is required for connecting servers and clients. One good approach to this kind of system is found in the V-system. It has a uniform I/O interface that can be used to connect processes [Cheriton87]. To properly integrate itself into the V distributed name space, however, each server must handle naming operations as well. The pseudo-device interface is simpler in this respect as the kernel takes care of naming.

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 4. 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. At 1024 bytes only one request fits into the server's request buffer, but there are extra context switches due to a bug in the scheduler.

Byte-stream mechanisms, such as UNIX pipes, UNIX System V named pipes [Dolotta80], UNIX pseudo-terminals, and 4.3 BSD sockets, are limited to providing a reliable byte stream between processes. Usually any extra processing, like terminal line editing or a network protocol, is implemented inside the kernel. Pseudo-devices move the processing out of the kernel, and allow for more general operations via iocontrol().

The Version Eight stream facility [Ritchie84] also 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 iocontrol() calls into special messages that appear in the byte stream to the server process. Another extension lets the server ''mount'' a stream on a file and return new streams to clients that open the file. This achieves the name transparency that pseudo-devices have, and lets the server multiplex itself among several clients. The main difference between pseudo-devices and the stream facility (aside from the underlying implementation) is the way the interfaces are used; the stream facility is designed to support different combinations of kernel-resident processing units, whereas the pseudo-device mechanism is oriented solely towards user-level implementation of services.

The watchdog facility proposed by Bershad and Pinkerton [Bershad88] provides a different way to extend the UNIX file system, but it can be used to achieve nearly the same effect as pseudo-devices. 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. The main differences between the systems are that pseudo-devices provide the server with an asynchronous read-write interface to reduce overhead, and the watchdog process can handle subsets of file operations.

7. Conclusion

Pseudo-devices provide a way to integrate a user-level server process into Sprite's distributed file system. By making the service appear as a special I/O device, the existing open-close-read-write interface is retained. Pseudo-devices are named and protected just like other files. Byte stream communication is via read() and write(), and read-ahead and write-behind can be used for asynchronous communication. Iocontrol() is available for operations specific to the pseudo-device, and can be used as the transport mechanism for an RPC system. The standard interface also means that the implementation of any specific pseudo-device could be moved into the kernel for better performance, or even implemented in hardware, without having to change any clients.

The performance of the implementation is acceptable at this point, although some further tuning may be possible. The buffering system represents our initial attempt to improve the performance and robustness of the system over an earlier pipe-based implementation. There is additional mapping overhead associated with copying data from one user process to another, so that pseudo-devices are not quite as fast as regular Sprite pipes. However, the request buffer is relatively simple for the server to manage, and the fact that it is pre-allocated allows some optimizations in the server.

We currently use pseudo-devices to implement terminal drivers, the X window server, and the TCP/IP protocol family. Future work includes extending the pseudo-device mechanism to a ''pseudo-file system'' mechanism that can be used to transparently access foreign file systems.

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.

Cheriton87. D. R. Cheriton, ''UIO: A uniform I/O interface for distributed systems'', *ACM Trans. on Computer Systems 5*, 1 (Feb. 1987), 12-46.

Dolotta80. T. A. Dolotta, S. B. Olsson and A. G. Petrucelli, *Unix User's Manual, Release 3.0*, Bell Laboratories, Murray Hill, NJ, June 1980.

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.

IP81. J. Postel, ''Internet Protocol'', *RFC 791*, Sep. 1981.

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.

Ritchie84. D. Ritchie, ''A Stream Input-Output System'', *The Bell System Technical Journal 63*, 8 Part 2 (Oct. 1984), 1897-1910.

Scheifler86. R. W. Scheifler and J. Gettys, ''The X Window System'', *ACM Trans. on Graphics 5*, 2 (Apr. 1986), 79-109.

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.