A Comparison of the Vnode and Sprite

File System Architectures

Brent Welch
welch@parc.xerox.com
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

This paper compares the vnode architecture found in SunOS with the internal file system interfaces used in the Sprite distributed file system implementation. The emphasis of the comparison is on generalized support for remote access to file system resources, which include peripheral devices and IPC communication channels as well as regular files. A strong separation of the internal naming and I/O interfaces is exploited in Sprite to easily provide remote access to all of these resources. In contrast, the vnode interface mixes naming and I/O operations in a single interface, making generalized remote access more awkward.

[A version of this paper appeared in the Proceedings of the USENIX File System Workshop, held in May, 1992, pages 29-44. This version has been submitted to Computer Systems.]

1 Introduction

Kernel support for remote file systems is centered around internal file system interfaces that hide the details of accessing file systems whether they are local or remote. Perhaps the most widely known of these interfaces is the SunOS vnode interface [Sandberg85][NFS85], although Ultrix has a similar gnode interface[Rodrigeuz86], and ATT UNIX has a remote inode interface[Rifkin86]. These interfaces are all evolutions of the original UNIX file system implementation in order to support remote file systems. The basic approach was to abstract the file and directory access procedures so there could be different implementations corresponding to different sorts of file systems. Examples include the original ATT UNIX local file system, the BSD fast file system format, and remote file systems such as NFS, AFS, and RFS. As well as introducing a generic interface, the original inode data structure was replaced with a more opaque object descriptor, which is called a *vnode* in SunOS, that represents objects implemented by different file systems.

The main property of all these designs is that they are oriented mainly towards file access, and this bias leads to a fundamental problem: naming and I/O operations are lumped together into the same internal file system interface. For example, there are operations on vnodes that locate their name in a directory, as well as operations that transfer data to the object represented by that name. By grouping these two different classes of operations into the same interface, the option of implementing pathnames by dedicated name servers is virtually precluded. Instead, the server for an object must implement both naming and I/O operations on its objects. For example, RFS provides remote device access, but you have to mount the /dev directory of the remote machine in order to access its devices. While the grouping of naming and I/O is reasonable for files, it is less convenient for peripheral devices and server processes, especially in an environment of diskless workstations.

In contrast to the designs that evolved from a UNIX implementation, the Sprite file system architecture provides generalized remote access to different kinds of file system resources that include peripheral devices and IPC channels as well as files and directories. The goal of the Sprite architecture is to use the distributed file system as a name server for objects other than files, and to do this is such a way that does not require a local file system on each workstation. The key to achieving this is to have two distinct internal file system interfaces, one for pathname operations and one for I/O operations. This makes it possible to have a pathname implemented by Server A that corresponds to a peripheral device on Client X, or an IPC channel to a process on Client Y.* The main point is that by properly separating the naming and I/O interfaces it is quite feasible to reuse the file server's directory structures as a global name space for any object accessed via the UNIX open-close-read-write interface.

The remainder of the paper discusses the vnode and Sprite architectures in more detail. Notable features of the Sprite architecture include a prefix caching system that replaces the UNIX mount mechanism, support for process migration and crash recovery, and an object-oriented design that has two base classes: pathnames and I/O objects.

2 The Vnode Architecture

The vnode architecture has two internal interfaces, the vnode and vfs interfaces. The vfs interface is concerned with the mount mechanism, while the vnode interface is concerned with access to objects within a file system.

The operations in the vnode interface are listed in Table 1. This is the interface as defined for SunOS 4.1.1. The details of each operation are not crucial to the arguments presented here, although some operations will be discussed in more detail below. The main thing to note that there is a rough classification of the operations into two sets. The first set of operations deal with pathnames: vn_access, vn_lookup, vn_create, vn_remove, vn_link, vn_rename, vn_mkdir, vn_rmdir, vn_readdir, vn_symlink, and vn_readlink. The other set of operations apply to the underlying object being named by a pathname (e.g., a file or a device). These operations include vn_open, vn_close, vn_rdwr, vn_ioctl, vn_select, vn_getattr, vn_setattr, vn_fsync, vn_lockctl, vn_getpage, vn_putpage, and vn_map. There is also a set of routines that deal more with the management of the vnode data structures themselves: vn_inactive, vn_fid, vn_dump, vn_cmp, vn_realvp, vn_cntl.

The vfs interface operations are given in Table 2. This interface is primarily concerned with the process of mounting a file system into the global directory hierarchy. The mount mechanism assembles self-contained directory structures on different disks (unfortunately called "file systems") into a single hierarchy. Each file system has a root directory. One file system is the distinguished root of the overall hierarchy. The vfs_mount operation is used to mount the root directory of other file systems onto an existing directory. Ordinarily file systems are mounted onto top-level directories of the root file system (e.g., /usr or /vol1), but it is legal to mount file systems on any directory, even one in a mounted file system (e.g., if disk 1 is mounted on /a, then disk 2 could be mounted on /a/b, or even /a/b/c).

*. Generally, the terms "server" and "client" refer to hosts and their operating system kernel.

Table 1. Vnode Operations <sys/vnode.h>

Operation	Description
vn_open	Initialize an object for future I/O operations.
vn_close	Tear down the state of the I/O stream to the object.
vn_rdwr	Read or write data to the object.
vn_ioctl	Perform an object-specific operation.
vn_select	Poll an object for I/O readiness.
vn_getattr	Get the attributes of the object.
vn_setattr	Change the attributes of the object.
vn_access	Check access permissions on the object.
vn_lookup	Look for a name in a directory.
vn_create	Create a directory entry that references an object.
vn_remove	Remove a directory entry for an object.
vn_link	Make another directory entry for an existing object.
vn_rename	Change the directory name for an object.
vn_mkdir	Create a directory.
vn_rmdir	Remove a directory.
vn_readdir	Read the contents of a directory.
vn_symlink	Create a symbolic link.
vn_readlink	Return the contents of a symbolic link.
vn_fsync	Force modified object data to disk.
vn_inactive	Mark a vnode descriptor as unused so it can be uncached.
vn_lockctl	Lock or unlock an object for user-level synchronization.
vn_fid	Return the handle, or file ID, associated with the object.
vn_getpage	Read a page from the object.
vn_putpage	Write a page to an object.
vn_map	Map an object into user memory.
vn_dump	Dump information about the object for debugging.
vn_cmp	Compare vnodes to see if they refer to the same object.
vn_realvp	Map to the real object descriptor.
vn_cntl	Query the capabilities of the object's supporting file system.

Table 2. Vfs Operations <sys/vfs.h>

Operation	Description
vfs_mount	Mount a file system onto a directory.
vfs_unmount	Unmount a file system.
vfs_root	Return the root vnode descriptor for a file system.
vfs_statfs	Get file system statistics.
vfs_sync	Flush modified file system buffers to safe storage.
vfs_vget	Map from a file ID to a vnode data structure.
vfs_mountroot	Special mount of the root file system.
vfs_swapvp	Return a vnode for a swap area.

There are two problems with the vnode interface design that limit its ability to provide generalized remote access and to share the file system uniformly throughout a network. One problem is that each host defines its own mount table. This stems from UNIX's past as a single-host, timesharing system. The other problem is that the vnode interface includes both naming and I/O operations. This stems from the file-oriented bias of the vnode architecture. Both problems are related to the way the pathname resolution algorithm was extended to a distributed environment.

In a stand-alone system, pathname resolution involves processing a pathname one component at a time by looking for that component in the current directory of the search. Each directory is checked to see if it is a mount point for another file system. If it is, the current directory of the search shifts to the root of the mounted file system. Symbolic links are also expanded during this process. Thus, this basic algorithm is a component-by-component traversal of a pathname, including expansion of symbolic links and indirections at mount points used to glue together file systems on different disks.

In a distributed system there are a number of places to split the pathname resolution algorithm between clients and servers. LOCUS put the split at the block access level so that remote pathnames were resolved by reading remote directory blocks over the network [Walker83]. The vnode architecture puts the split at the component access level. The vn_lookup operation takes a directory handle and pathname component as arguments and returns the handle on the named file, if it is found. Similarly, the contents of symbolic links are retrieved through the vn_readlink call. Sprite and RFS put the split at the pathname access level so that component-by-component iteration happens on the server. The appeal of a lower-level split is that there is less effect on existing higher-level code. The advantage of a higher-level split is that more functionality can be moved to the server and the number of client-server interactions can be reduced. Note that the vnode architecture tries to optimize by keeping a cache of recent name translations. However, an NFS server cannot guarantee the consistency of this cache, so each entry for remote files remains valid for only a few seconds on the client. In general, a higher-level interface hides more details from the client and gives more flexibility to the server.

The vnode design also requires that each client maintain its own mount table so that its lookup procedure can do the indirection at mount points. As a system gets larger, the job of keeping all the clients' mount tables consistent becomes a problem. SunOS introduced an automounter mechanism to work around the problem of mount table consistency. The basic idea of an automounter is that a user process maintains a map from mount points to file systems. The automounter maps are kept in a distributed database service, NIS. The automounter process postpones the mounting of file systems by mounting itself onto top level mount points. When an access is made to a "file system" mounted on the automounter process, the automounter does the real mount and returns in such a way that the operation is retried. The point of delaying the kernel-level mounts is to increase availability. First, using a file system served by a crashed server can hang a process in many NFS implementations, so keeping as few NFS file systems mounted as possible reduces the chance of accidentally hanging on a crashed server.* Second, if a file system is replicated, then postponing the mount increases the chance that the file system will be mounted from a working server. Thus, by retaining the mount mechanism, a new automounting mechanism has to be introduced to keep mount tables consistent across clients.

So far it seems like the problems in the vnode interface can be fixed up. However, perhaps the biggest problem is the mixing of naming and I/O operations in the vnode interface. The mixture is easy to fall into from the viewpoint of a simple file system of regular files and directories. Directories are just special cases of files that name other files. Both objects are resident on the same disk, so the same host can be the server for both, and the same internal data structure (vnode) can be used for both. But consider a peripheral device. First, there are often many different names that map to the same device. With tape drives, for example, different names imply different tape densities and whether or not the tape is rewound when it is closed. This creates the following problem. The result of pathname resolution is a vnode that corresponds to the special file that names the device. In order to perform I/O on

*. Computing the current working directory often results in accesses to other file systems, and many shells do this computation quite often. A recursive algorithm scans the contents of the parent directory and makes a stat() call on every entry in order to figure out the name of the current directory. If there are any directories with mount points for multiple servers, then the crash of any of the servers can hang the algorithm.

the device, another data structure, the *snode* that corresponds one-to-one with the device, is needed to serialize operations and maintain state about the device. Under the covers, the special device file system implementation must maintain snodes and arrange for all the vnodes associated with device names to have a pointer to the real object descriptor, the snode. The vn_realvp operation in the vnode interface is used to map from vnodes to snodes.

Again, there seems to be a fix. But what about remote device access? The mixture of naming and I/O operations implies that if you want to provide access to a peripheral device then you also need to export its name. This distributes the responsibility for naming among all servers. In UNIX terms, it means that you have to mount the /dev directory of a remote host into your file system in order to access its peripheral devices. Even a diskless workstation needs its own file system that has a /dev directory with names for its peripheral devices. This forces all clients to maintain a file system, which increases administrative overhead. It can also increase cost and noise if it means adding local disks.

Forcing each host to have its own file system may not seem like such a bad thing. After all, a workstation might be able to function with its own file system if the file servers are unavailable. However, in practice most workstations are configured so that they depend heavily on file servers, in spite of local storage. Only the bare minimum is on the local file systems, and most important files, including system programs, are on remote servers. This approach is taken to reduce administration effort. In such an environment, why is it necessary to have a local file system at all?

What about extending the vnode architecture to a multicomputer where the hosts are even more tightly bound than in workstation environments? Clearly it should not be necessary to maintain 1024 nearly identical private file systems just so each node can mount together file systems. On the other hand, if file system operations are forwarded to distinguished file server nodes, then the lookup traffic in the root directory becomes a bottleneck. What is needed is a system that efficiently partitions the name space among different servers, and then separates naming and I/O so that inter-node device access does not involve file server nodes.

In summary, the vnode and related architectures result in a world of many remote file systems that happen to be accessible. In contrast, the Sprite architecture provides a single file system view that transparently incorporates the resources available on all machines without highlighting machine boundaries.

3 The Sprite Architecture

The Sprite file system architecture presented here is a second-generation design that was implemented after initial experience with Sprite, which we began building in 1985. The current design has been operational since 1989. The goal of the architecture is to generalize the remote access capabilities that support remote file access to also accommodate remote device and remote IPC access. In Sprite, remote access also involves state management to support process migration and failure recovery, not just I/O. The architecture starts by cleanly separating naming and I/O operations so that file servers can easily act as name servers for devices and inter-process communication (IPC) channels. It also eliminates the need for private client file systems, and it makes a different cut between client and server in pathname operations that is more efficient than the component-level split in the vnode interface.

As with the vnode design, the Sprite design uses a generalized *object descriptor*, which is a main-memory data structure maintained by the kernel, not a disk-resident representation of an object. A basic object descriptor has a type, uid, server ID (a Sprite Host ID), a reference count, and a lock bit. Objects are specified internally by a tuple of <type, serverID, uid>. This base data structure is subclassed* for the implementations of various object types. The object-oriented approach allows clean separation of different object implementations, as well as sharing between similar objects like remote devices and remote pipes. A diagram of the file system architecture is given in Figure 1.

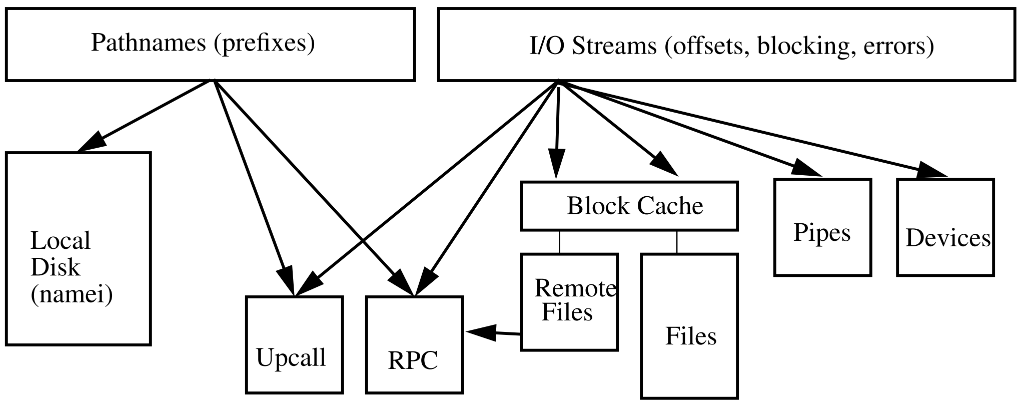

Figure 1. An overview of the Sprite file system architecture. The two primary interfaces involve pathnames and I/O streams. The Upcall module forwards operations to user-level processes. RPC forwards operations across the network to other Sprite kernels.

The pathname interface illustrates the three basic cases handled by the Sprite kernel. The server for a pathname may be the local kernel, in which case the file system implementation is accessed by an ordinary procedure call within the Sprite kernel. The server may be remote, in which case a kernel-to-kernel RPC protocol is used to pass the pathname to the server [Welch 86a]. Finally, the server may be a user-level process, in which case an upcall mechanism, which was described in [Welch 88], is used to pass the operation up to a user-level *pseudo-device* or *pseudo-file-system* server process. Thus there are three orthogonal cases that are supported by a Sprite kernel, a local, kernel-resident module, a remote module, and a user-level module.

3.1 Separating Naming and I/O

Consider the UNIX open system call that maps from a pathname to an I/O stream. In Sprite, this is broken into two operations, NameOpen and IoOopen, that involve both of the internal file system interfaces. The NameOpen procedure returns attributes of the named object. The IoOpen procedure uses these attributes to create an open I/O stream. The NameOpen and IoOpen procedures may be implemented by different servers, and this is achieved cleanly by branching through the object-oriented naming and I/O interfaces.

The clean separation of naming and I/O means that objects like devices can rely on a file server to implement the naming interface on their behalf. In Sprite, special files are used to

*. The various kinds of object descriptors would be the result of subclassing if Sprite were written in C++. However, all the object-oriented features described here were hand crafted in C.

represent devices and pseudo-devices in the name space. Pseudo-devices can be considered a kind of IPC channel [welch 88]. Furthermore, a Sprite file server can have special files that represent devices and pseudo-devices on any machine in the network. Contrast this with NFS, which doesn't support remote device access, or even RFS, which only supports accesses to devices on the file server. Those systems are limited by the vnode (or equivalent) interface that lumps naming and I/O together.

The vnode architecture has a "cloning" mechanism that is used to achieve a similar effect of having distinct NameOpen and IoOpen procedures. The vnode returned from the vn_open call can be cloned to get better handle on the underlying object. For example, this is when the snode that corresponds to a device is located and linked back to the vnode. However, this is a client-side operation that is invoked after the server does the lookup, and it is oriented towards the special case of special device files.

The Sprite naming and I/O interfaces are described in Tables 3 and 4.

Table 3. Sprite Naming Interface <fs/fsNameOps.h>

Operation	Description
NameOpen	Map from a pathname to attributes of an object, and prepare for I/O.
GetAttributes	Map from a pathname to attributes of an object.
SetAttributes	Update the attributes of an object.
MakeDevice	Create a special file that represents a device.
MakeDirectory	Create a directory.
Remove	Remove a named object.
RemoveDirectory	Remove an empty directory.
HardLink	Create another name for an existing object.
Rename	Change the pathname of an existing object.
SymLink	Create a symbolic link.

Table 4. Sprite I/O Interface <fsio/fsio.h>

Operation	Description
IoOpen	Complete the preparation for I/O to an object.
Read	Read data from the object.
Write	Write data to the object.
PageRead	Read a page from a swap file.
PageWrite	Write a page to a swap file.
BlockCopy	Copy a block of a swap file. Used during process creation.
IoControl	Perform an object-specific operation.
Select	Poll an object for readability, writability, or an exceptional condition.
IoGetAttributes	Fetch attributes that are maintained at the object.
IoSetAttributes	Change attributes that are maintained at the object.
ClientVerify	Verify a remote client's request and map to local object descriptor.
Release	Release references after an I/O stream migrates away.
MigEnd	Acquire new references as an I/O stream migrates to a host.
SrvMigrate	Update state on the I/O server to reflect an I/O stream migration.
Reopen	Recover state after a server crash.
Scavenge	Garbage collect object descriptors.
ClientKill	Clean up state associated with a client.
Close	Clean up state associated with I/O to an object.

Most of the operations in the two interfaces are self explanatory. The Page and Block operations are for use by the VM system when managing swapping files. The main reason these routines are distinct from the ordinary Read and Write routines is that the caching strategy is different for swap files. Clients only cache pages for read-only segments (code and initialized data), while all swap pages are cached on the server. The BlockCopy routine is used to duplicate swap files on the server during process creation.

Table 5 lists the different types of object descriptors. A comparison with Figure 1 shows there are more kinds of descriptors than there are modules in the block diagram. This reflects sharing among the implementations of the various remote cases. Most of the remote cases share their implementations of the Read, Write, Ioctl, and Select operations, which are just RPC stubs. The RPC stubs rely on the host ID in the object descriptor ID in order to direct their RPC requests. The state-related operations (e.g., IoOpen, Release, MigEnd, Close) differ among the various cases, although often not greatly. The remote file implementation is the most specialized remote case because it is optimized to use the local cache. In the case of a cache miss it uses the same RPC routines as the other cases.

There are corresponding local and remote descriptors for various kinds of objects like files, pipes, devices, and pseudo-devices. This reflects the need to maintain different state information on clients and servers. There is, however, a well defined mapping between a client's object descriptor ID and the corresponding descriptor on the server; the IDs differ only in the type field. The RPC stubs on the server use the ClientVerify routine to map the ID, fetch the server's descriptor, and verify that the client is known to the server. The operation is then passed through the I/O interface so that all object implementations are accessible remotely. The importance of this approach is that it extends the general features implemented in upper levels of the kernel to local, remote, and user-provided objects. Notable, high-level features include the name space, error recovery, and blocking I/O. Thus, the focus of the file system architecture has relatively little to do with actual disk management. The focus is on extending the high-level abstractions of pathnames and I/O streams to the network environment.

Table 5. Sprite Object Descriptor Types <fsio/fsio.h>

Type	Description
LCL_FILE	A file, directory, or symbolic link stored locally.
RMT_FILE	For a remote file, directory, or link.
LCL_DEVICE	A device on this host.
RMT_DEVICE	A device on a remote host.
LCL_PIPE	An anonymous pipe buffered on this host.
RMT_PIPE	An anonymous pipe buffered on a remote host.
PDEV_CONTROL	Used by a pseudo-device server to listen for connection requests.
SERVER	The upcall channel to a pseudo-device server.
LCL_PSEUDO	Client's handle on a local upcall channel.
RMT_PSEUDO	Client's handle on a remote upcall channel.
PFS_CONTROL	Used by a pseudo-file-system server to listen for connections.
PFS_NAMING	Upcall channel used for naming operations in a pseudo-file-system.
LCL_PFS	Client's handle on local upcall channel to pfs server.
RMT_PFS	Client's handle on remote upcall channel to pfs server.
RMT_CONTROL	Used during get/set I/O attrs if pseudo-device server is remote.
PASSING	Used to pass existing I/O streams from a pdev server to a client.

3.2 Integrating User-Level Servers

Pseudo-devices are user-level processes that implement the I/O interface by way of an upcall mechanism [welch88]. They are used to implement the X server, terminal emulators, and a TCP/IP server. Unlike UNIX ptys that use pairs of special device files, there is a single pathname that represents a pseudo-device in Sprite. This pathname is implemented as a kind of special file in a directory on a Sprite file server. In this case, the user-level server only implements the I/O interface and the file server implements the pathname interface for the pseudo-device. The user-level server process opens the file and can listen for connections by client processes. When a client opens the pseudo-device file, a new upcall channel is created for communication, and the server process gets a new open file descriptor to represent it.

A pseudo-file-system is a whole subtree of the file system that is implemented by a user-level server process. This is used to implement a gateway to NFS file systems. In this case a prefix is registered in the local prefix table, and pathname operations that match on that prefix are forwarded up to the user-level server process over an upcall channel. Prefix tables are described in more detail in Section 4. An open operation on a pathname in a pseudo-file-system results in a new upcall channel for I/O operations on the named object.

The upcall implementation has a number of object descriptor types, although there are really only 4 different kinds of object descriptors used. Extra types were introduced to handle special cases, sort of a poor man's subclassing mechanism. For each pseudo-device there is one control descriptor that keeps state about where the user-level server process is executing. The control descriptor is also used by the server to listen for client connections. 2 descriptors are used to represent the client and server sides of an upcall channel. The passing descriptor is used to return an open I/O stream in response to a pseudo-file-system open request, which is an alternative to creating an upcall channel. This could be used to open files or devices via a pseudo-file-system, although this is still not fully implemented and debugged.

The Plan-9 system also integrates user-level server processes into its name space. It uses the mount system call to attach a full-duplex pipe to a name. All client operations, both naming and I/O, from all clients, are passed through this channel along with a unique tag so the server can manage requests. The main difference from Sprite pseudo-device and pseudo-file-systems is that the Sprite kernel maintains separate channels corresponding to each client open system call, and in the case of pseudo-devices it implements the naming interface on behalf of the server. There are also optimizations in the Sprite implementation to allow for asynchronous reads and writes between the client and server.

3.3 Handling Special Files

A final touch is required to implement support for special files cleanly on the file server. After a file server finds a file, it needs to take type-specific action to complete the servicing of a NameOpen request. This special action is what distinguishes a NameOpen operation from a GetAttributes operation. Another, single-procedure interface called the SrvOpen interface is used to abstract these type-specific actions. The implementation of SrvOpen is selected based on a type field in the disk-resident descriptor for a file (the disk inode in UNIX terms). Currently there are three different implementations of the SrvOpen procedure.

The first implementation is used for regular files, directories, and symbolic links. This procedure invokes the Sprite cache consistency algorithm [Nelson 88] and sets up enough state about the remote client so that the IoOpen procedure does not need to contact the file server a second time. This is an important optimization for the common case of file access.

A second implementation is used for device files. In this case the procedure just has to extract the relevant attributes from the disk-resident descriptor that are needed by the remote client. Sprite device files have a ServerID attribute in addition to the standard UNIX major and minor device numbers. The ServerID identifies what host controls the device (i.e., the I/O server). A simple trick is played here to allow sharing of the /dev directory. A special value of the ServerID is mapped to the host ID of the client making the NameOpen RPC, thus mapping the special device file to the instance of the device on that client. This trick makes it easy to have a single /dev directory that is shared among all the hosts in the Sprite network. Most entries in /dev are these generic device files that map to the local instance. Some entries have specific ServerID attributes so they map to devices (e.g., tape drives) on particular hosts. Finally, note that this arrangement results in a full many-to-many mapping from pathnames to objects that really requires a clean separation of the naming and I/O interfaces.

The third implementation of SrvOpen is used for pseudo-devices. The file server maintains state about which host the pseudo-device server process is executing on, and it updates and verifies this state when a pseudo-device is opened. Server processes supply an extra flag on the open call to distinguish themselves, and after that one or more clients can open the pseudo-device and get a connection to the server.

4 The Sprite Prefix Table Mechanism

Sprite prefix tables were originally described in [Welch 86b], but the following description is included for completeness.

Sprite uses a prefix table mechanism to implement a uniformly shared, hierarchical name space. Each Sprite kernel keeps a cache of pathname prefixes. The prefixes define the way server domains are coalesced into a single hierarchy, and their use completely replaces the UNIX mount mechanism. The Sprite naming protocol ensures that servers export their domains consistently so that all hosts, and therefore all processes, see exactly the same name space. Users, administrators, and developers enjoy the simplicity of a single, shared name space. The fully shared file system supports cross-compilation and easy maintenance for all architectures from any workstation.

In contrast, the V-system [Cheriton87] and Mach 3.0 use a prefix cache that is maintained on a per-process basis by library routines. While this is advertised as a feature that allows custom name spaces, I believe this is a case where generality is not what you want. I think that the real reason V and Mach provide per-process prefix caches is because they are implemented in the run-time library instead of the kernel. Plan-9 [Pike90] relies heavily on per-process name spaces created by mounting various servers into the name space. Sprite also mounts services into the name space, even user-level processes as described below, but again, the name space is global and shared by all processes.

The basic idea of prefix caching is simple. On a client, the longest matching prefix selects the *domain* for the pathname. An object ID is registered with the prefix that identifies the server for the domain, the domain's type (e.g., local, remote, or user-level), and a uid that identifies the domain to the server. The server is sent the part of the pathname after the prefix, along with the object ID. Relative pathnames bypass the prefix match and are sent to the server of the current working directory, along with the object ID for the current directory. Note that the interface to the server is the same in both cases. In summary, clients match prefixes to choose servers. Servers process pathnames relative to one of their domains. When servers completely resolve a pathname, they perform the requested pathname operation (NameOpen, Remove, Rename, MakeDirectory, etc.).

4.1 Pathnames that Involve Multiple Servers

The main complication with prefix matching is that a pathname can wander through many different server domains, so the initial prefix match may not lead to the right server. The solution to this problem is to return control to the client when a pathname leaves a server's domain. This gives the client an opportunity to cache information about the next server, including a prefix that may optimize future pathname operations.

A pathname exits a domain when it encounters a symbolic link to an absolute pathname, or if it specifies “..” in the domain’s root directory. In these cases, the server returns a new pathname to the client. The returned pathname is either the new absolute pathname resulting from the symbolic link expansion, or a pathname that begins with “..”. The client combines the latter form with the prefix used to select the server in order to get a new absolute pathname. The “..” and the last component of the prefix are removed so the pathname will match on a different prefix.

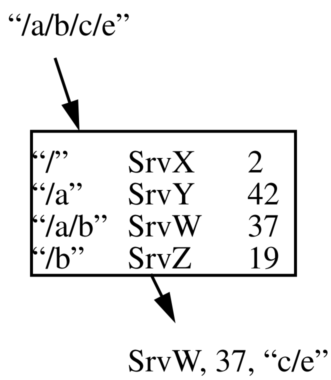

Figure 2a: Prefix match on the client selects longest prefix, “/a/b”.

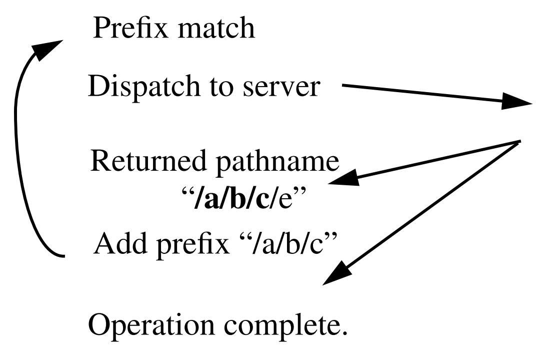

Figure 2b. Lookup iteration on the client faults in new prefix, “/a/b/c”.

Mount points are handled by placing a special symbolic link at the mount point called a *remote link*. Mount points can occur in any directory, so it is possible to nest server domains arbitrarily. The contents of a remote link is the prefix, or absolute pathname, of the mount point. The link has a different file type than ordinary symbolic links so that the server knows when it hits a mount point. The server expands the remote link and returns the new pathname to the client. In addition, the server indicates of how much of the returned pathname is the prefix of the mount point so that the client can add the prefix to its cache. This is illustrated in Figure 2b. Note that there is nothing in the remote link but its own name, so some other mechanism is used to locate the server for that domain. Currently, Sprite uses broadcast to locate the server. After locating the server and obtaining the object ID for the root directory of the domain, the client reiterates the lookup procedure. On the next iteration the returned pathname will match on the new prefix, and the lookup will be directed to the next server.

The iteration over the prefix table is contained within two routines on the client, one for single pathname operations and one for two pathname operations. Higher-level procedures such as Fs_Open or Fs_Rename (these implement the open and rename system calls) package up their non-pathname parameters into a structure and call the appropriate prefix table routine. The arguments to the call are the pathname(s), an operation identifier (e.g., for NameOpen or Rename), the structure that collects the remaining miscellaneous arguments, and a structure for the return values of the operation. The prefix table routines do the prefix match, branch through an operation table depending on the server’s type and the requested operation, and handle the case where pathnames are returned from the server by reiterating the procedure. In the call to the server, the pathname arguments are replaced by the object ID and relative pathname that result from the prefix match.

Bootstrapping is achieved by broadcasting for the server of “/”. Initially all lookups are directed to the root server. Remote links will cause prefixes to be returned to the client. As a client’s prefix cache fills up, the prefix matches usually direct operations to the correct server, bypassing the servers for higher levels of the directory hierarchy.

Measurements of the Berkeley Sprite network revealed that about 17% of pathnames suffered redirections back from a server, but less than 1% of the pathnames were redirected because they hit a remote link [Welch90]. Instead, ordinary symbolic links between domains caused the bulk of the redirections. It should be possible to extend the prefix caches to cache the results of absolute symbolic link expansion so that these redirections are eliminated, but this has not been implemented.

4.2 Operations on Two Pathnames

The Rename and HardLink operations involve two pathnames, and they are constrained to operate on pathnames in the same domain. The problem is that the two pathnames may or may not start out in the same domain, and they may or may not end up in the same domain. The required iteration over the prefix matches is described below. Note that in the best case only a single server operation is required. By making Rename, and HardLink, into a single server operation, the server can easily implement these atomically without having to maintain state between multiple client operations.

The client matches both pathnames against the prefix table and obtains object IDs and relative pathnames for both. It sends these to the server identified by the object ID for the first pathname. The server traverses the first pathname. If the pathname leaves its domain, it returns a new pathname to the client as described above. In this case, the client reiterates the prefix matches and retries the operation. If the first pathname terminates in the server’s domain, then it proceeds to traverse the second pathname. If the second pathname also termintates in the same domain, the server performs the requested operation. Otherwise, if the second pathname exits the domain, or does not even start in it, then the server returns EXDEV to indicate a potential conflict. At this point the client has to verify the conflict because it is still possible for the second pathname to end up back in the same domain as the first one. To verify the conflict, the client does a GetAttributes of the parent directory of the second pathanme. The parent is checked to avoid file-not-found errors. If the GetAttributes results in a returned pathanme, then the Rename or Hardlink operation is retried. Eventually the iteration terminates with a successful server operation or a verification that the two pathnames are in different domains.

4.3 Pros and Cons

There are a number of good properties of the Sprite prefix mechanism, and one limitation. First, clients are simplified. They do not iterate through directories or expand symbolic links, in contrast to the vnode architecture. The prefix mechanism completely replaces the UNIX mount mechanism, so servers are no more complex. The interface is optimized so that system calls usually require only a single server operation. The most important property is that the name space remains uniform across machines because it is the contents of the symbolic links at the mount points that defines how domains fit together, not a per-host or per-process configuration file. An final advantage with the prefix caching mechanism is that the root server is usually bypassed because clients quickly cache prefixes for the domains they use.

The primary limitation of the Sprite scheme is the use of broadcast to locate servers. This choice was made for simplicity, but it obviously limits the range of the name space. A general solution would be make an upcall in the case that the broadcast fails so that a user-level process can take arbitrary action to locate the server. For example, the Domain name server or some other name service could be used. This solution has the nice property that the kernel implements a lightweight mechanism (broadcast) that works for the common case, but can rely on the escape hatch to user-level in the hard case.

5 Maintaining State in a Distributed System

Another way to view the differences between the vnode and Sprite architectures is their support for maintaining distributed state. The Sprite architecture is inherently stateful, in contrast to the stateless file server model used in NFS, which was the main reason the vnode architecture was introduced. Sprite's stateful nature originally stemmed from its cache consistency mechanism that supports data caching on diskless workstations [Nelson88]. Delayed writes are used to optimize performance, but stateful servers keep track of how files are cached so they can guarantee UNIX file access semantics even when files are concurrently write shared by processes on different hosts. It turns out that server state is useful for a variety of other things. Examples include the record of which host is executing a pseudo-device server, what files are open for execution so they cannot be overwritten, what devices are open in case they wish to enforce exclusive access, and file locking protocols.

The Sprite I/O interface has a number of entry points with no counterparts in the vnode interface, and these are almost entirely devoted to maintaining distributed state. The state does not have to be complex, it just has to be maintained carefully. It boils down to a per-object client list that records a few words of state about how that client is using the object. For most objects all that is required is the number open I/O streams to an object, expanded into counts for readers, writers, and executors. Shared and exclusive lock state is also kept in the client list, as well as type-specific state such as the current version number of files, or the ID of the host executing a pseudo-device server. For efficiency, a summary of the client-list information (e.g., the total number of readers) is also kept in the server's object descriptor.

Maintaining state during normal operations is simple. State is initialized by the IoOpen and/or SrvOpen procedures. When handling RPC requests, the ClientVerify procedure is invoked to ensure that the server knows about the client. IoClose cleans up state about an I/O stream. This approach implies that the I/O server is contacted as a result of each open and close system call. However, I/O stream sharing that results from process creation does not require contact with the I/O server. This is an important optimization, and relatively easy to achieve by keeping a reference count on steam descriptors ("file descriptors" in UNIX terminology).

Life gets complicated by crash recovery and process migration [Douglis91]. A complete description of these mechanisms can be found in [Welch 90]. Only a few key points about these mechanisms will be made here.

5.1 Process Migration

Process migration results in open I/O streams that move around the network. This requires coordinated state updates on the original client, the new client, and the I/O server. This is complicated by multiple references to I/O streams that operate independently, yet share a common stream access position, or offset. When a process migrates, state about each of its I/O streams is packaged up and shipped to the process's new site as part of the process descriptor. However, no state changes are made when the process begins to leave. Instead, after the process is reincarnated on the new site a set of coordinated state changes occur. The new site notifies the I/O server for each stream, indicating that a stream reference has migrated to it. The I/O server makes a callback to the original client that retrieves the current stream offset and decrements the stream reference count there. The I/O server then updates its client list and replies to the new site. If there are no remaining stream references at the original site, then the new site can own the stream offset. Otherwise the server maintains the stream offset on behalf of both sites. This algorithm is somewhat delicate because there are often Close operations that occur about the same time as migrations, and the locking order on data structures must be deadlock free.

5.2 Crash Recovery

Crash recovery is based on the following observations. First, the I/O servers keep their state organized on a per-client basis in order to support things like data cache consistency. This also makes it easy to clean up state about clients that crash. Second, it turns out to be straight-forward for clients to mirror the state that the server keeps about them. Recall that the state information is just counts of open I/O streams, plus type-specific state like file version numbers. The observation that clients can duplicate their server's state means that the system can recover from server crashes. The Reopen entry in the I/O interface is an idempotent operation that attempts to reconcile the client's state with the state that exists on the server. The success or failure of the Reopen call is dependent on the type of the underlying object and the actions of other clients. For example, a client can recover if it is writing to a file and has dirty blocks in its main memory cache, unless for some reason (i.e., a network partition) the server has allowed a different client to open the file and generate a conflicting version. Two features of this recovery scheme are worth repeating. First, the Reopen is idempotent so the client can invoke it whenever it thinks the server's state is out of date. This happens when the client gets an ESTALE return code that implies the server has forgotten about the client, or when the client senses a server reboot as described below. Second, the server has the final word, and can always deny a reopen request.

Crash detection is an important part of the recovery process. Sprite kernels monitor their RPC traffic to other hosts and keep a state variable for each other host. The state can be one of Booting, Alive, or Dead. Transitions between these states are used to trigger recovery actions by clients and servers. When a server assumes a client is in the Dead state, it cleans up state about that client. When a client assumes a server is newly Alive, it initiates its recovery protocol. Of course, crash detection is not fully reliable because a host can really only detect a communication failure. It is for this reason that the recovery protocol is idempotent. A server might prematurely clean up state about a client, and a client might prematurely initiate the recovery protocol.

The Sprite RPC protocol has two features that aid crash and reboot detection. First, a timestamp is part of the RPC packet header, and this is set to the time the host booted up. All that is important is that it is different on each boot so clients can detect a server reboot even if they didn't try to contact it while it was down. Second, an "I'm booting" flag is also part of the RPC packet header. This flag is cleared after a server is fully up (e.g., has checked its disks) and is ready for service. Finally, idle clients ping their server every 30 seconds to make sure they detect reboots in a timely way. A server could have thousands of clients before the load from this would be a problem.

6 Other Related Architectures

The Ultrix gnode interface [Rodriguez86] is quite close to the vnode interface. The rnode interface used in ATT Unix, however, shares some similarities with the Sprite architecture.- The RFS architects classify it as a "remote system call" interface [Rifkin86]. Remote operations are trapped out at a relatively high level and forwarded to the remote node. For pathname operations in particular, this can result in fewer client-server interactions than a component-based interface. The implementation is a little gory, however. System calls are littered with checks against the remote case, and on the server side longjmp is used to warp execution back to the RPC stubs. This was done in order to avoid deeper structural changes required for a fully modular implementation. In contrast, the Sprite implementation is quite clean, which made it easy to add in new cases such as the notion of user-level servers.

The UIO interface of the V system is a clean design introduced to support Uniform I/O access in distributed systems[Cheriton87]. It is also coupled with a prefix table mechanism that is used to partition the name space among servers [Cheriton89]. However, the UIO interface also lumps naming and I/O operations together into one interface. The prefixes on pathnames are used to partition the name space among different classes of servers such as print, file, display, and tape servers. A global name service is used to map from prefixes to server multicast addresses, but each class of service still has to implement a name space for its objects. In contrast, the Sprite prefix mechanism transparently distributes a hierarchical name space among file servers, and the file system interfaces are designed to allow the file servers to function as more general name servers.

The Echo distributed file system developed at DEC SRC uses a more elaborate name service to transparently distribute a file hierarchy among servers. The system is complicated by support for replication, both at the name server level and at the file server level [Hisgen89]. In Echo, hosts still have local file systems, and the global file system name space is not used for remote device or remote service access.

The Plan-9 architecture [Pike 90], like Sprite, uses the file system name space to represent services, both kernel-resident and user-level. The details of the communication mechanisms are different, and the Sprite name space is fully shared via the prefix table mechanism, while the Plan-9 name space is per-process-group via the mount mechanism. Remote access is possible in Plan-9 by means of a server-server that can make connections to remote services. Sprite has additional mechanisms to support process migration and server crash recovery.

7 What I Would Do Differently

Perhaps the weakest point in the Sprite design is the handling of object attributes. Currently this is distributed between the file server and the I/O server for the object. For files those servers are the same, but for devices and pseudo-devices they are different. Both the UNIX stat() and fstat() system calls result in RPCs to both the file server, to get most of the attributes, and the I/O server, to update attributes like the modify time. Of course this is optimized for regular files so there is just one RPC. Also, the access and modify time stamps on the generic device files, the pathnames that map to local device instances, are not that well defined.

There are some more minor things that could be cleaned up. It turns out that the Release and Close operations are very similar, as are the IoOpen and MigEnd procedures. These operations maintain usage counts that reflect the number of read and write streams to an object. These four procedures could probably be replaced by a pair of slightly more general procedures, one to add a new stream, and one to remove state about a stream.

Finally, there are still some holes in the implementation that reflect lack of programming cycles. For example, while it is possible to migrate the client of a pseudo-device, migrating the server is a lot harder because the state of all the clients also needs to be updated. For some pseudo-device servers, like the X server, it doesn't really make sense to migrate the process to another host. With others, like the daemon that maintains state about idle hosts, it might be useful.

A more interesting missing feature is the ability of a pseudo-device or pseudo-file-system server to return a previously existing I/O stream in response to the IoOpen operation. Currently the IoOpen procedure only creates an Upcall channel to the pseudo-device server. However, the ability to pass back arbitrary streams would make it possible to implement archive file systems and version control file systems. The infrastructure that supports process migration makes it feasible to pass an I/O stream between any two processes, but integration with the pseudo-device IoOpen procedure just wasn't on any critical path so it didn't get fully implemented.

Finally, the decision to have a private kernel-to-kernel RPC protocol is often questioned by others. The pat answer is that pseudo-devices and the associated upcall mechanism provide IPC, albeit with an open-close-read-write file system model. It can be argued that ioctl() is a general transport mechanism, that hierarchical pathnames are well suited for server names, and the model benefits from being transparently distributed. In practice the file system model has worked fine for basic client-server relationships such as the X server and its windowing clients. However, someone interested in more general protocols would probably prefer a pure message passing system. Our decision to hide the network protocol reflects the goal of managing the network on behalf of users by providing a familiar time-sharing-like environment centered around the file system.

8 Conclusions

The main point that this paper makes is on the importance of cleanly separating the naming and I/O interfaces of the file system. This split is taken for granted in classical distributed systems literature that always includes a system wide name server [Wilkes80]. However, the distributed systems that evolved from UNIX implementations failed to incorporate this notion in the right way. SunOS, for example, uses a network database server to name hosts, password entries, and maps from file systems to file servers. However, once one enters the domain of a file system all things are tied to one host. Even distributed systems like V use a name service to partition servers at the top levels of the naming hierarchy. In Sprite the file servers generalize their directory structure mechanism to provide naming support for a variety of objects. The only things not named by the file system are hosts, which can be named via the Domain Name Service, users, and arbitrary processes (only pseudo-device servers have names in the file system name space).

9 References

Baker91. M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, "Measurements of a Distributed File System", *Proc. of the 13th Symp. on Operating System Prin., Operating Systems Review*, Oct. 1991, 198-212

Cheriton87 D. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems", *ACM Transactions on Computer Systems* 5, 1 (Feb. 1987), 12-46.

Cheriton89 D. Cheriton and T. Mann, "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance", *ACM Transactions on Computer Systems* 7, 2 (May 1989), 147-183.

Clark85. D. Clark, "The Structuring of Systems Using Upcalls", *Proc. of the 10th Symp. on Operating System Prin., Operating Systems Review* 19, 5 (Dec. 1985), 171-180.

Douglis90. F. Douglis, "Transparent Process Migration for Personal Workstations", PhD Thesis, Sep. 1990. University of California, Berkeley.

Hisgen89 A. Hisgen, A. Birrell, T. Mann, M. Schroeder and G. Swart, "Availability and Consistency Trade-offs in the Echo Distributed File System", *Proc. Second Workshop on Workstation Operating Systems,* Sep. 1989, 49-54

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

Pike90 Rob Pike, Dave Presotto, Ken Thompson, and Howard Tricky, "Plan 9 from Bell Labs", Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, 1-9.

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

Walker83 B. Walker, "The LOCUS Distributed Operating System", *Proc. of the 9th Symp. on Operating System Prin*., Operating Systems Review 17, 5 (Nov.1983), 49-70

Welch86a. B. B. Welch, "The Sprite Remote Procedure Call System", Technical Report UCB/Computer Science Dpt. 86/302, University of California, Berkeley, June 1986.

Welch86b. B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", *Proc. of the 6th ICDCS*, May 1986, 184-189.

Welch88. B. B. Welch and J. K. Ousterhout, "Pseudo-Devices: User-Level Extensions to the Sprite File System", *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

Welch90. B. B. Welch, "Naming, State Management, and User-Level Extensions in the Sprite Distributed File System", PhD Thesis, 1990. University of California, Berkeley.

Wilkes80 M. Wilkes, R. Needham, "The Cambridge Model Distributed System", Operating Systems Review 14, 1 (Jan. 1980).

10 Appendix A - Summary of Remote Procedure Calls

There are 40 RPCs defined in the Sprite implementation, of which 29 are for the file system, 4 are for process migration, 2 are for remote signals, 1 is to get the current time, and 4 are for testing the RPC system itself. 2 of the file system RPCs are directly in support of process migration. Recall that the RPC mechanism is entirely kernel-to-kernel. The operating system uses RPC for its own purposes in order to provide a transparently distributed system. The large number of file system RPCs reflects the degree to which Sprite relies on its distributed file system to provide network transparency.

Table 6. Kernel-to-kernel RPCS used in Sprite.

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