A Comparison of three Distributed File System Architectures: Vnode, Sprite, and Plan 9

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

Abstract

This paper compares three distributed file system architectures: the vnode architecture found in SunOS, the architecture used in the Sprite distributed file system, and the architecture used in the Plan 9 distributed file system. 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. The vnode architecture is an evolution of structures and interfaces used in the original, stand-alone UNIX file system. The Sprite architecture provides a network-wide shared name space and emphasizes a strong separation of the internal naming and I/O interfaces to easily provide remote access to a variety of resources. The Plan 9 architecture relies on per-process name spaces and is organized around a single file system protocol, 9P, through which a variety of devices and system services are accessed.

1 Introduction

This paper compares three approaches to distributed file system architecture. The Vnode (or VFS) architecture is part of SunOS. It was added to the original Berkeley version of UNIX (4.1BSD) as that system was adapted to support networks of Sun workstations [Sandberg85][NFS85]. The Sprite file system was built as part of the Sprite network operating system project [Ousterhout88]. Sprite was designed and built from scratch to support a networked environment of high performance, multi-processor workstations [Hill88] and associated servers. Plan 9 is the latest version of the operating system built at ATT Bell Labs [Pike90], the lab at which the original version of UNIX was written [Ritchie74]. To admit a bias up-front, it should be noted that the author was part of the team that designed and built Sprite [Welch90].

There has been considerable work in distributed file systems, and these three systems certainly do not encompass every approach taken. However, they share enough features that a comparison is interesting. Many basics are similar and are derived from these systems' UNIX heritage. File system objects are named in a hierarchical name space that is distributed among multiple servers. The interface to the file system is in terms of open, close, read, and write operations. Different kinds of objects such as peripheral devices and user-provided server processes are accessed via the file system interface, as well as ordinary files. Thus, on the surface these systems are quite similar, but there are a number of interesting differences in their internal structure that reflect different approaches to providing a similar high-level interface in a distributed environment.

Each of these systems takes a simple object-oriented approach that takes into account the variety of system resources accessed via the file system interface. Each system defines an internal file system interface, or set of interfaces, that hides the differences among the resource types such as files and devices. Associated with the different resources are typed object descriptors that encapsulate the state for an instance of the resource. Given this similar approach to architecture, we can compare these systems by looking at the operations and data structures associated with their main internal file system interfaces. After an overview of each architecture, more detailed comparisons of particular features will be presented.

The terms "client" and "server" are used frequently in this paper. In nearly all cases they refer to instances of an operating system acting in one of these two roles. Typically a client operating system is acting on behalf of a user program that has made a system call requiring access to a remote resource. Typically a server is the operating system kernel that has direct access to a given resource such as a file or peripheral device. However, there is also discussion of user programs that export a service via the file system interface, and these situations will typically be clarified with the term "user-level server".

2 The Original UNIX Architecture

A brief description of key aspects of the original UNIX architecture will serve as a starting point for the comparisons. There are three aspects of the system introduced here: composition of the name space, name lookup, and I/O accesses.

The name space for the UNIX file system is a hierarchy of directories and leaf nodes. A leaf node can represent a disk-resident file, a peripheral device, a communication port, or another directory, which leads to a hierarchical name space. The top of the hierarchy is called the root directory, or just "the root". The hierarchy is divided into subtrees (unfortunately termed "file systems") that are self-contained hierarchies. One subtree is taken to be the overall root of the hierarchy. Other subtrees are attached by the *mount* operation that associates the root of a subtree with an existing directory in the overall hierarchy.

Name lookup proceeds by recursively traversing the directory hierarchy. A lookup starts at the root with an *absolute pathname*, or at the process's current directory with a *relative pathname*. A pathname is represented as a sequence of directory names separated by slash characters ('/') and it ends with the name of a leaf node or a directory. An absolute pathname is distinguished because it begins with a "/", which is the name of the root directory. Lookup is a simple matter of scanning the current directory for the next component in the pathname. A match advances the current directory or completes the lookup, and a miss terminates the lookup with an error. The only complications are support for the mount operation and for *symbolic links*. A table of mount points is maintained by the operating system, and this is checked at each component to see if the lookup should indirect to the root of a different subtree. A symbolic link is leaf node that contains a pathname. The system substitutes the value of the symbolic link for the current component and continues the lookup operation.

The cost of name lookup is amortized over many I/O operations by using the *open* operation to create a data structure that is used by subsequent *read* and *write* operations to access the underlying object directly. The read and write operations also mask differences among file system resources so that general purpose programs can be used to access a variety of data sources.

The following sections give a brief overview of each of the three systems being considered. More detailed comparisons among the systems will be given after the overviews.

3 The Vnode Architecture

The approach taken in SunOS with its vnode (or VFS) architecture is to retain the basic structure found in the original UNIX system, but to generalize it by introducing two internal interfaces that can have multiple implementations under them to support different kinds of file systems. The *vfs* interface is concerned with the mount mechanism and allows for different types of file systems (e.g., local or remote) to be mounted into the hierarchy. The vfs operations are given in Table 1. This is the interface as defined for SunOS 4.1.1.

Table 1. 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.

Table 2. 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.

The *vnode* interface is concerned with access to individual objects. The operations in the vnode interface are listed in Table 2. There is a rough classification of the vnode operations into three 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 second 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 choice of operations in the vnode interface reflects an evolutionary design choice to handle network distribution. The original directory scanning lookup algorithm is retained, which means that the main loop of pathname resolution is done locally, and vnode operations are performed to find a name in a given directory and to retrieve the value of a symbolic link. This structure means that many remote vnode operations may be required to resolve a pathname. The mount mechanism is retained, which implies that each host must manage its own set of mount points in order to assemble its file system from those available on remote file servers. Thus this architecture can be characterized as a modification to a stand-alone system that allows for remote access.

The central data structure in the architecture is the vnode that represents a particular object. There are also per-file system structures associated with the vfs interface. The vnode contains some generic fields for data structure locking, a vector of vnode operations, a pointer to the vfs structure, and then some object-specific pointers. There is both an opaque data pointer, as well as a union of three pointers to either data pages, a streams data structure [Ritchie84], or a socket data structure. This set of pointers indicates that streams and sockets are not fully integrated into the architecture. Instead of doing all stream and socket operations through the vnode operations, there are many places in the code in which special case checks are made against the streams case, and fewer places where the socket case is checked explicitly. This approach defeats some of the goals of the vnode interface which should be to abstract the differences among object types.

4 The Sprite Architecture

The Sprite file system architecture was designed to support a network of diskless workstations served by a collection of file servers. There are two aspects of its design that reflect a network-orientation as opposed to an evolution of a stand-alone design. The first of these is a shared, network-wide name space that reflects a goal of the operating system to manage distribution on behalf of users and application programmers. All hosts have identical views of the system, so they can be used just like terminals on a timesharing system. The second aspect of the design is a separation of the internal pathname operations and I/O operations into different interfaces. The internal split between naming and I/O reflects a goal to reuse the capabilities of the file servers as more general name servers.

The shared name space is achieved by replacing the mount mechanism with a system based on prefix tables[Welch86]. The prefix tables divide name lookup into a prefix matching part, which is done on clients, and a directory scanning part, which is done on the servers. There are two key properties of the Sprite prefix table system. The first is that the naming protocol between clients and servers causes new prefixes to be returned to clients as they access new parts of the shared name space. The second is that clients dynamically locate servers when they add new prefixes to their cache. These properties mean that managing the distributed system is easy. The system administrator configures the servers to export their file systems under particular prefixes, and the naming protocol automatically propagates the prefix information to clients.

The internal interfaces used in Sprite separate operations into those on pathnames, which involve the prefix table mechanism, and those on open I/O streams. An I/O stream can be hooked to a file, peripheral device, or to a user-level server process. The operations in the two interfaces are given in Tables 3 and 4. The distinction between the two interfaces provides the flexibility to implement names on the file servers for resources that are located on the client workstations. There is no need for client workstations to maintain a directory structure, either on disk or fabricated in main memory data structures, in order to access their local resources. They do, however, depend on the file servers being up, but so do workstations in most UNIX networks. At the same time, the shared name space and division of the naming and I/O interfaces automatically extends the ability to access the peripheral devices and service processes on other workstations.

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 cached object descriptors.
ClientKill	Clean up state associated with a client.
Close	Clean up state associated with I/O to an object.

The central data structure in the Sprite file system is an *object descriptor*, analogous to the vnode. The basic object descriptor has a type, UID, a reference count, some flags and a lock bit used to serialize access. This base structure is embedded into more complete object descriptors that correspond to the different types of objects accessed via the file system. Above the naming and I/O interfaces only the base structure is accessed, with the more specific structure remaining private to the various object implementations.

The client-side and server-side object descriptors for the same object are different. The client's descriptor is often very simple, just containing enough state to support the recovery protocol described in Section 12, while the server's object descriptor contains all the information needed to directly access the object. Special support for the remote case is provided in two ways. First, the UID explicitly encodes the server ID so that remote operations can be easily forwarded over the network. Secondly, remote operations only pass types and UIDs, not object descriptors. A mapping between local and remote types is used on the server to find the object descriptor that corresponds to the client's object type and UID. This mapping is done by the ClientVerify operation, which is selected by the client's object type, but which returns the server's object descriptor.

5 The Plan 9 Architecture

The architecture for the Plan 9 file system is centered around a single protocol named 9P and the use of per-process name spaces. The 9P protocol reflects a design goal in which special case operations are eliminated by treating nearly all resources as file systems. That is, a resource can have a name space associated with it, and the resource is manipulated by writing and reading messages to the various names associated with the resource*. Per-process name spaces allow customization that is used to support heterogenous clients and virtual device environments for the 8 1/2 window system.

The operations in 9P are listed in Table 5. They can be grouped into four categories, session management, authentication, naming, and I/O. The nop, session, and flush messages are used to condition the communication channel to a server. A channel is a full duplex byte stream that is used to multiplex different client requests, and a session is established with a server for a particular sequence of client requests. The auth and attach messages are used to obtain a new, authenticated channel to a server. The channels starts out logically attached to the root of the server's tree. The clone, walk, and clwalk operations are used for pathname traversal. Clone duplicates a channel, and walk moves the channel to a different named object. Clwalk combines these operations. Finally there are the operations on individual objects: open, read, write, clunk (i.e., close), create, remove, stat and wstat.

Table 5. Plan 9 Protocol, 9P.

Operation	Description
Nop	The null operation.
Session	Start a communication session.
Error	Error return message in response to other messages.
Flush	Abort an outstanding request message.
Auth	Authenticate a client.
Attach	Identifies the user and the server name space to be accessed via the channel.
Clone	Duplicate a reference to an object.
Clunk	Close a reference to an object.
ClWalk	Combines the Clone and Walk operations.
Open	Prepare an object reference for future I/O operations.
Create	Create an object and prepare for I/O on it.
Read	Read data from an object.
Write	Write data to an object.
Remove	Remove an object from a server.
Stat	Fetch the attributes of an object.
Wstat	Write the attributes of an object.
Walk	Descend a directory hierarchy.

*. The few important resources not modeled as file systems in Plan 9 include process creation and virtual memory. Creating a process is important and complex enough to warrant a special purpose implementation. Virtual memory is not modeled as a file system because of the complexity associated with distributed shared memory implementations.

The 9P protocol specifies objects in two ways, with client-specified *fid*s and server-specified *qid*s. An fid is 16 bits and is chosen by the client to represent a particular reference to some object. The qid is a 64 bit unique ID for an object chosen by the server. For example, in the attach operation, the client specifies an fid that will represent its reference to the server's root directory. The clone operation specifies a new fid that represents another reference, and then the walk operation is used to change the association between the fid and a server object. The qid is returned from the attach, open, create, and walk operations in order to uniquely identify the underlying object.

In addition to the 9P protocol, there are two system calls that are used to manipulate the per-process name spaces. *Mount* attaches a communication channel to a name, and *bind* gives a new name for an existing object. If bind is used on directories, the contents of the new directory can be merged with the existing contents, forming a *union directory*. One of the parameters to bind specifies a search order in the case of conflicting names. Union directories are used to replace the search path mechanisms used in other UNIX systems. Ordinarily a process shares its name space with the rest of its process group. However, a new process can also get a copy of the name space and use mount and bind to customize the name space.

The following subsections consider particular aspects of a distributed file system implementation and how they are effected by the design choices of the three systems.

6 Remote Access

Remote access is a fundamental aspect of a distributed file system, so the communication protocols used for remote access are an important part of the system architecture.

The vnode architecture makes no particular commitment to remote access. Instead, it is possible that some file system types support remote access via their own protocol (e.g. NFS[NFS85], AFS[Howard87] or RFS[Rifkin86]). In addition, a mount protocol is used to obtain a handle on a remote file system. While it is not fair to blame the vnode architecture for the faults of NFS, it is important to note that the full vnode interface is not exported via NFS. The subset does not include open, close, or ioctl operations. This changes the semantics that can be offered by remote file access because the server must be "stateless". For example, the server does not remember that a file is opened for execution, so it will allow the file to be overwritten, which usually causes an error when new pages of the file are mixed with old pages as a result of demand paging. Also, NFS is oriented towards file access; it is not general enough for remote device access.

In Plan 9, the file system interface is defined as a set of messages over a communication channel, so it extends quite easily to a remote environment. A kernel-based service called the mount device is used to attach communication channels to remote servers into the name space. The mount device converts system calls on the remote resource into 9P messages on the communication channel. The message format is fairly simple, beginning with an operation type and a call sequence number and followed by operation-specific arguments. One thing to note is that there can be multiple clients that use the same channel to a server, and the server is responsible for managing all the requests.

Sprite uses a kernel-to-kernel Remote Procedure Call (RPC) protocol for its network communication [Welch86]. The goal in using kernel-to-kernel RPC is to provide an efficient means of remote communication that provides structured semantics. Sprite RPC is layered over ethernet packets directly, or encapsulated in IP packets. The current Sprite implementation defines 40 RPCs, 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, which is described in detail by Douglis [Douglis90]. Most of the file system RPCs correspond directly to procedures in the two main file system interfaces. In addition, RPC is used for cache consistency callbacks, and a broadcast form of RPC is used to support server location. Note that while most RPCs are issued by client workstations and serviced by file server hosts, all Sprite hosts are capable of servicing RPC requests. For example, cache consistency callbacks are issued by the file server and serviced by a workstation that caches the file. Process migration involves RPCS between workstations. The complete set of Sprite RPCS is listed in Table 6.

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.

7 Pathname Resolution

In a distributed system there are a number of places to split the pathname resolution algorithm between clients and servers. Early systems put the split at a low level so they could preserve existing higher-level code. Early versions of LOCUS modified UNIX and put in remote access at the block access level so that remote pathnames were resolved by reading remote directory blocks over the network [Walker83]. Sun originally used a network disk device in a similar manner to support its diskless workstations. The primary limitation of a remote block access protocol concerns sharing. LOCUS used a complex replication scheme to coordinate updates to shared blocks, while Sun's network disk device did not support read-write sharing at all.

The next level for the interface is at the component access level. This is done in the vnode and Plan 9 architectures. This structure keeps the main lookup loop on the client side. If pathnames are mostly served by remote servers, then lookup will be expensive because it requires a remote operation for each remote component of the pathname. Also, high-level file system operations like open and rename require multiple calls through the file system interface. The lookup function (e.g., vn_lookup or walk) is used to obtain a handle on a file, and then a second function is applied to that object (e.g., vn_remove or remove).

Sprite (and RFS [Rifkin86]) put the split at the pathname access level so that component-by-component iteration happens on the server. 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. Complex operations like create or rename can be done in a single server operation. For example, the Fs_Rename and Fs_HardLink operations in Sprite send both pathnames to the server so it can perform the operation atomically. This eliminates the need for the server to maintain state between client operations.

The Sprite prefix table system works as follows. The local operating system (or "client") matches pathnames against a cache of pathname prefixes. The longest matching entry in the cache determines the server for the pathname, and the remaining part of the pathname is sent to the server for processing. Ordinarily the server can process the complete pathname and perform the requested operation. However, it might be the case that the pathname leaves the subtree implemented by the server. Servers detect this by placing a special kind of symbolic link at points where other subtrees are attached (i.e., mount points). The contents of such a *remote link* is its own absolute pathname, which is the prefix for names below that point. When a server encounters a remote link, it combines the link value with the remaining pathname to create a new pathname. The new pathname is returned to the client, along with an indication of what part of the pathname is the valid prefix.

If a new prefix is returned to the client as the result of a naming operation, then the client broadcasts to locate the server for that prefix. Once the server is located, or if it is already known, then the client restarts the lookup. At this point the pathname matches on the new, longer prefix, and the lookup is directed to the proper server. The process is bootstrapped on a client by broadcasting for the server of the root directory. Initially all requests are directed to the root server, but soon the prefixes for other file system domains are cached by the client and the root server is bypassed. The iteration between client and server is also used to correct stale entries in the client caches.

The iteration over the prefix cache is slightly more complex for the operations that involve two pathnames, Fs_Rename and Fs_HardLink. The main idea is that in the best case only a single server operation is required.

Caching can be applied to these systems to speed name resolution. Caching of lookup results and file attributes is used in the vnode implementation to eliminate some remote accesses. However, the stateless NFS servers do not support caching well so the clients must discard their cached data relatively quickly, reducing the effectiveness of their caches. In Sprite, only prefixes are cached, so each name resolution requires at least one server operation. This makes it easy for Sprite servers to keep track of file usage in order to support the file data caching system described below. Plan 9 does not cache name lookup results. The only optimization for remote clients in Plan 9 is the clwalk operation that combines the clone and walk operations. This is designed for clients accessing the file system over a slow link such as a 9600 baud phone connection.

8 Name Space Management

The main issue with managing the name space is keeping it consistent so that programs can be run on any host in the network without worrying about irregularities in the name space. If consistency is not automatically provided, the burden of managing the name space is pushed onto the workstation users, who probably will not succeed in keeping their workstation consistent, or onto the system administrator, where the effort grows with the size of the system.

The vnode architecture does not provide any explicit support for consistency, so an external mechanism called an automounter is introduced. The automounter consults a network database to find out how file systems are to be mounted into the overall hierarchy. It introduces a level of indirection into name lookup that lets it monitor access to the remote file systems, and to mount and unmount those file systems in response to demand. The indirection is achieved by having the automounter process implement a directory full of symbolic links into another directory that contains real mount points. For example, the automounter mounts itself onto "/net". In response to a lookup of "/net/sage", the automounter makes sure that the corresponding remote file system is mounted in an alternate location (e.g., "/tmp_mnt/net/sage"). Its response to the lookup indicates that "/net/sage" is a symbolic link. This triggers a second call on the automounter to retrieve the value of the link. Finally the lookup gets routed to the remote file system by the new pathname "/tmp_mnt/net/sage". Note that the automounter process remains involved in the lookup of every pathname beginning with "/net" because it provides the implementation of the symbolic link from there into the real file system.

Plan 9 supports consistency by providing a standard template for the per-process name spaces, and by sharing the name space among members of a process group. The file "/lib/-namespace" consists of a set of mount and bind commands that assemble the prototypical name space. This file is used by init, the first user process, and it is consulted by other commands such as the one that executes programs on remote hosts. The commands in "/lib/-namespace" can be parameterized with environment variables to take into account different users and machine types. Additionally, users can supply their own set of mount and bind commands in a personal namespace file, "/usr/$user/lib/namespace".

Sprite builds consistency into the name lookup algorithm so that the name space is shared consistently among hosts. There are two configuration steps that need to be taken by the system administrator to add a new file system subtree. First, the server is configured to export the subtree under a given prefix. Second, a remote link is created so the server of the parent subtree knows where the mount point is. For example, if "/a/b/c" is a newly exported subtree, then the remote link "c" in the directory "/a/b" is created with the value "/a/b/c". The presence of this link triggers broadcasts by clients to locate the server for "/a/b/c". The range of this search could be extended by adopting internet multicast instead of broadcast.

These approaches to name space consistency can be classified as an external solution, a solution by convention, and an internal solution. The external automounter solution used in SunOS is basically a retrofit to fix a problem in their initial system. Its implementation is clever, but less efficient than a built in solution. The solutions used in Plan 9 and Sprite acknowledge the consistency problem up front and were built into those systems from the start.

9 Naming and Accessing Devices and User-level Servers

This section compares the way these systems allow extension of the distributed file system to name and access other kinds of objects. The notion of extending the file system name space to include other objects such as peripheral devices has been around for a long time [Fiertag71].

Naming can be a tricky issue because of a UNIX tradition of giving the same device a number of names that correspond to different uses of the device. The best (or worst) example is that tape devices use different names to specify the density of the tape media and whether or not to rewind the tape when the device is closed. The opposite problem is that a single name (e.g., /dev/console) could be used on different hosts to refer to different devices (i.e., the local instance of the device). Another example is a user-level server that is known by a single name, except that each client that binds to the service (i.e., opens its name) gets a private communication channel to the server. The general property that these scenarios share is that the name of the object and the object itself are in fact different things, and, in general, might be implemented by different servers.

The vnode architecture support access to non-file objects. Peripheral devices are named with special files that record the type and instance of the device (i.e., its *major* and *minor* numbers). UNIX domain sockets are communication channels (pipes) that have names in the file system, so they can essentially be used to name user-level servers. However, both of these mechanisms predate the vnode architecture, and they were not extended to provide transparent remote access to devices and servers. For remote access, programs must use TCP or UDP socket connections which are chosen out of a different name space and created with a different set of system calls than open(). The NFS protocol does not support device access because it lacks open, close, and ioctl. In spite of its limitations, NFS has been used in a number of systems to provide interesting extensions [Minnich93][Gifford91]. If the vnode architecture included a general way to export the vnode interface to user-level then these projects might have been even easier, and others, such as remote device access, could have been added.

Plan 9 extends its name space in a clean fashion by allowing servers, user-level or kernel-resident, to mount themselves onto names in the name space. The servers handle all the 9P protocol requests for names below that point. Many servers implement small name spaces to represent different control points. Remote access to these services is provided via an *import* operation that makes a portion of a remote name space visible in the local name space. The decision about how to import remote name spaces is concentrated into a few different programs, and most processes just inherit their name space from their process group. In practice, this results in partial sharing of remote name spaces in order to access particular devices such as the mouse and keyboard of a terminal for a process running on a compute server. This is a fine point that contrasts the per-process name space, which requires explicit sharing arrangements, with a network-wide shared name space.

Sprite uses the global file system name space to name devices and user-level servers, so these can be named and accessed from any host in the network. The split between the internal naming and I/O interfaces allows the name to implemented as a special file on a file server, while the device or service is implemented on a different host. The attributes of the special file indicate on which host the device or service is resident. A trick is employed in order to present UNIX programs with a single directory, "/dev", that has the names for commonly used devices. A special value of the location attribute maps to the host of the process opening the device. Thus, device files can represent devices on specific hosts, or they can represent devices on the local host.

User-level extensibility in Sprite is provided in a modular way by providing an upcall mechanism that forwards kernel operations up to a user level server process. The split means that either the naming interface or the I/O interface, or both, can be exported to user-level. For example, a Sprite file server can implement the name for the X display server running on a workstation. Clients of the window system access the display server using the file system interface. Open goes through the file server that implements the name (e.g., /hosts/sage/X0), while read and write calls are handled by the display server. As an other example, a CASE tool could provide a different name space for files stored on a regular Sprite file server*.

10 Blocking I/O

Blocking I/O operations become more important when the file system is used to access peripheral devices and server processes that might take an arbitrary amount of time to respond with data. UNIX adds an additional source of complexity with the select system call that can be used to wait on a set of I/O streams to be ready for I/O. Additionally, I/O streams can be set into non-blocking mode so that if a read or write is attempted when no data is ready, a special error code is returned.

There are basically two choices regarding blocking I/O operations: blocking at the server or blocking at the client. The vnode and Plan 9 architectures include no special support for long term blocking, so the operations implicitly block at the server. This means that servers have to manage blocking operations, and they have to decode whether requests should be non-blocking or not.

In Sprite, long term blocking is done on the client. Servers return a special error code if they cannot process a request immediately, and they keep a list of tokens that represent waiting processes. They support the select operation in a similar way. When a device becomes ready for I/O, the server notifies any waiters. The notification procedure will use RPC in the case that the token represents a process on a remote host. On the client, the main read and write routines are constructed as loops around calls through the I/O interface to the type-specific read or write routine. The loop checks for the blocking return code, and it also

---

*. The ability to open a kernel-supported object via a user-level server is not fully implemented in Sprite. It leverages off the mechanisms used by process migration to migrate open I/O streams between workstations. The migration mechanisms work fine, but their integration into the user-level open procedure is not complete. As such, the only user-level servers that provide a name space (e.g., an NFS gateway) also provide I/O access to their objects.

handles any races with notifications from the server. The advantage of this approach is that the interface to the server is always the same, regardless of the blocking mode set for the client's I/O stream. Also, the same notification mechanism is used to support blocking I/O and the select system call, so select can be applied to a set of I/O streams no matter where the servers for the I/O streams are located.

11 Data Caching

Data caching is known to be an effective way to optimize file system performance. The file system manages a cache of data blocks in main memory in order to eliminate disk accesses. This section discusses the way the different architectures handle data caching in a distributed environment where caching can eliminate network traffic as well. The main issue with distributed caching is *consistency*. That is, how to network sharing patterns and the caching mechanism interact? If a cache always returns the most recently written data, regardless of sharing patterns, then it is regarded as a consistent cache. A closely related issue concerns the writing policy of the cache. A few of the common policies include write-through, in which all data is written immediately through to the server; write-through-on-close, in which the write back is postponed until the file is closed; and delayed-write, in which the write is postponed for a delay period before being written.

The vnode architecture includes a cache, but it does not include an explicit cache consistency scheme. It is up to the network file systems below the interface to provide their own consistency scheme. Both NFS and AFS have different caching schemes.

NFS uses a scheme that relies on polling by the client and short time-outs on data to shorten windows of time in which inconsistent data could returned from the cache. It uses a write-through-on-close policy. When data is placed into the cache, it is considered valid for a short length of time, from 5 to 30 seconds depending on the type of data, recent accesses to the file, and the particular client-side NFS implementation. During this time period a client will use the cached data without verifying its modification time with the server.

AFS uses a callback scheme in which the servers tell clients when cached data is no longer valid. If a client is using a file for an extended period of time, it must periodically renew its callback registration. It also uses write-through-on-close. AFS maintains its cache on the local disk, and uses the ordinary UNIX block cache to keep frequently used data blocks in main memory [Howard88].

The Plan9 architecture has no explicit support for caching. There is one file system implementation that does caching for clients that access the file system over a slow link such as a 9600 baud phone connection. The caching file system does write-through, and it checks the validity of the cached data when each file is open. There is no server support in the way of callbacks.

Sprite supports caching through a cache consistency protocol between clients and servers. Like AFS, servers make callbacks when cached data is no longer valid. In addition, a callback may indicate to the client that it can no longer cache the file because of network sharing. In this case all I/O operations go straight through to the server and are serialized in the server's cache. Clients check the validity of cached data each time a file is opened, and they tell the server when the file is closed so the server can keep accurate account of which clients are using its files. The check at open-time, the server callbacks, and the ability to disable caching on a file-by-file basis allows the caching system to be perfectly consistent

[Nelson88]. Sprite uses delayed-write caching. This means that data ages in the cache for 30 seconds, even if the file is closed. This allows for reduction in network traffic if the file is temporary and gets deleted after use. A study of the effectiveness of the caching scheme is given in [Welch91] and [Baker91].

12 Crash Recovery

A distributed system raises the possibility that clients and servers can fail independently. When a client fails, the servers may need to clean up state associated with the client. When a server fails, the clients need some way to recover if they have any pending operations with the server.

In vnode-based systems the failure recovery issues are left up to the network file systems implementations. NFS takes a simple approach to failure. It relies on "stateless" servers that do not keep any long term volatile state about particular clients. This means there is no need for them to clean up if a client fails. In turn, the clients typically address server failures at the level of network RPC. The default behavior is for the clients to keep retrying an operation until the server comes back on line. This behavior has been extended to allow for time-outs after some number of attempts, and to allow the RPC to be interrupted by a signal.

AFS supports read-only server replication in order to improve availability. The AFS cache manager will fail over to another server if the primary server fails. Client cleanup is addressed by the time-outs on the server callback registrations. The server can delay service after booting for this period of time and know that clients do not expect any callbacks. If the server were to crash during a write operation, the client would get an error.

Plan9 does not currently have a mechanism to handle server failure in a graceful way. This limitation is being addressed in a new version of the system, but currently the failure of the root file server requires a reboot of the clients. Client cleanup is addressed by the use of sessions in the communication protocol with the servers. Clients establish new sessions after they boot up, and this gives the server a chance to clean up any state left over from previous sessions.

Sprite servers maintain state about their clients to support the cache consistency scheme. The state allows the server to clean up after a client fails. There is also an idempotent recovery protocol between clients and the server that allows the server's state to be reconstructed after it fails. The idempotent nature of the protocol means a client can invoke the protocol when it suspects the server's state is different that it expects, such as after a network partition. The recovery protocol is based on mirroring state on the clients and the server. The state reduces to a per-client count of the I/O streams to each of the server's objects, plus a few object-specific items such as the version number for a file. It is straight-forward for clients to keep the same state as the server keeps about that client. After the server reboots (or a network partition ends), its clients contact the server with their copy of the state information. The server checks for any inter-client conflicts, and then rebuilds its state.

A Sprite server has the right to deny recovery on an object-by-object basis because not all object types can support recovery. Tape drives, for example, typically get rewound upon system reset, making it difficult to continue client operations. Files are subject to a possible conflict due to client caching. It is possible that the current writer of a file can be partitioned from the server by a network router failure. In this case the server might allow a different client to write a new version of the file. When the original client attempts to recover its state, that file will be in conflict.

13 Other Related Architectures

The Apollo DOMAIN system was probably the first commercially successful remote file system. It was modeled as a single-level-store system in which the virtual memory system was used to map files into address spaces. The system was also based around an object-oriented interface that provided extensibility [Leach82][Leach83].

The LOCUS system was another early distributed UNIX system. It started as a modification to UNIX that put in remote access at the disk block level, although the remote interface was gradually moved higher up in the system to more like the vnode interface. LOCUS includes support for replication, and parts of this system have been incorporated into versions of AIX [Popek85][Walker83].

The Ultrix gnode interface [Rodriguez86] is quite close to the vnode interface.

The rnode interface used in ATT Unix 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 not as clean, however; the remote case was added as a series of special cases in the code. This includes such tricks as patching the kernel-to-user copy routine to use RPC if needed. 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. Some of the issues in porting RFS into the vnode framework are described in [Chartock87].

The UIO interface of the V system is a clean design introduced to support Uniform I/O access in distributed systems[Cheriton87]. Like the Plan 9 protocol, 9P, the UIO interface lumps naming and I/O operations together into one interface, and the name space is managed on a per-process basis. It is also coupled with a prefix table mechanism that is used to partition the name space among servers [Cheriton89]. Unlike Sprite, the prefix-based naming scheme does not include an iterative mechanism that loads new prefixes into the client caches.

The Echo distributed file system developed at DEC SRC is an ambitious file system that includes support for replication at several levels: the name service, file service, and disk subsystem[Hisgen89]. The name service implements the upper levels of the name space hierarchy, and it uses a background propagation scheme for updates. File servers implement the lower parts of the name space and use a quorum-based voting scheme for updates in which a client can act as a "witness", or tie-breaking vote. 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.

14 Overall Comparisons

As a high-level, broad-bush comparison, I consider the vnode architecture to be a slight generalization of the stand-alone UNIX architecture that allows for remote file access. Many of the problems with remote access in a vnode-based system have more to do with the NFS protocol than with the architecture itself. AFS makes some improvements with respect to availability and caching support, but it does not address remote device access.

AFS did, however, add a general "vnode-intercept" layer in order to implement its cache manager in a user process, and this is similar in spirit to the support for user-level servers in Sprite. The Plan9 and Sprite architectures depart from the original UNIX architecture in more radical, although, different directions. Plan9 emphasizes a per-process name space and a single file system protocol, while Sprite has a network-wide shared name space and 2 main protocols, one for naming and one for I/O. Being complete redesigns, Plan 9 and Sprite offer better support for extensibility, that is, for making more things accessible via the file system interface. This includes remote access as well as integration of user-level server processes.

There are trade-offs between the per-process and network-global name spaces. Plan 9 takes advantage of the per-process name space in order to provide virtual environments for the purposes of remote execution, the window system, and to allow monitoring of the file system protocol. The potential pitfalls of managing many different name spaces are addressed by using a master template for the name spaces, and by concentrating the complexity of customizing the name space into a few programs. Sprite's shared name space is simple for users and makes it very easy to administer a network of workstations, most of which are diskless. The shared name space lead to a different approach to dealing with machine heterogeneity in which machine-dependent files are explicitly segregated into subdirectories that indicate their machine specificity. With respect to the window system, Sprite supports X, so there is no need for private device names. Furthermore, by accessing the X server via the file system interface, remote display servers are automatically accessible.

One distinct aspect of Sprite is its use of two primary internal interfaces, one for naming operations and one for I/O. This split is taken for granted in classical distributed systems literature that always includes a system wide name server [Wilkes80]. However, the distributed file systems that evolved from UNIX implementations tend to lump naming and I/O operations together. While the vnode architecture has a second interface for dealing with mounting the file system name space together, the main interface for object access has both naming and I/O operations. This reflects a heritage in which a host with disks implements files and directories together. However, a clean split between the two classes of operations allows the directory structure of the file servers to be reused to name other types of objects such as devices and user-level servers. This split also complements the use of the prefix caching scheme that transparently distributes the name space among servers. The prefix table system replaces the mount operation used in the vnode and Plan 9 systems, plus it keeps the global name space consistent.

15 Acknowledgments

Software architecture is a difficult thing to export from a project. I hope the readers can extract the important lessons learned that I learned while building Sprite, and can usefully extend those lessons to current and future distributed systems projects. I would like to thank Peter Honeyman for giving me the opportunity to present this material, first at the USENIX File System Workshop, and now in *Computer Systems*. I would like to thank Rob Pike for encouraging a more complete treatment of Plan 9 in this comparison, and for providing quick answers to probing questions. Most importantly, of course, I have to thank John Ousterhout for providing me the opportunity to work on Sprite, and the rest of the Sprite group for all their great efforts.

16 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

Chartock87 H. Chartock, “RFS in SunOS”, *USENIX Conference Proceedings*, Summer 1987, 281-290.

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.

Gifford91 D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O’Toole, Jr. “Semantic File Systems,” *Proc. of the 13th Symp. on Operating System Prin. Operating Systems Review*, 25 (5) (Oct 1991), 16-25

Hill86. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer* 19, 11 (November 1986).

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

Leach82 P. J. Leach, B. L. Stumpf, J. A. Hamilton and P. H. Levine, “UIDS as Internal Names in a Distributed File System”, *Proc of ACM SIGACT News-SIGOPS* Aug 1982, 34-41.

Leach83 P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson and B. L. Stumpf, “The Architecture of an Integrated Local Network”, *IEEE Journal on Selected Areas in Communications* SAC-1, 5 (Nov. 1993), 842-857.

Minnich93 R. G. Minnich, “The AutoCacher: A File Cache Which Operates at the NFS Level,” *Proc. of the Winter 1993 USENIX Conference*, (Jan 1993), 77-84

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.

Popek85 G. J. Popek and B. J. Walker, *The LOCUS Distributed System architecture*, MIT Press, Boston, 1985.

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

Rodriguez86 R. Rodriguez, M. Koehler, and R. Hyde, “The Generic File System”, *USENIX Conference Proceedings*, June 1986, 260-269

Sandberg85 R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, “Design and Implementation of the Sun Network Filesystem”, *USENIX Conference Proceedings*, June 1985, 119-130

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.

Welch91 B. B. Welch, "Measured Performance of Caching in the Sprite Network File System", *Computer Systems* 3, 4 (Summer 1991), 315-342

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