Spritely NFS: Experiments with Cache-Consistency Protocols

V. Srinivasan
University of Wisconsin

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory

Abstract

File caching is essential to good performance in a distributed system, especially as processor speeds and memory sizes continue to improve rapidly while disk latencies do not. Stateless-server systems, such as NFS, cannot properly manage client file caches. Stateful systems, such as Sprite, can use explicit cache consistency protocols to improve both cache consistency and overall performance.

By modifying NFS to use the Sprite cache consistency protocols, we isolate the effects of the consistency mechanism from the other features of Sprite. We find dramatic improvements on some, although not all, benchmarks, suggesting that an explicit cache consistency protocol is necessary for both correctness and good performance.

1. Introduction

Cache management strategies are central to performance, reliability, and correctness of distributed file services. Caching improves performance by avoiding unnecessary disk traffic, network traffic, and server use, but caching implies the potential existence of multiple copies of the same data, and keeping these multiple copies consistent is a challenge. This is especially true when the caches are kept by the clients of a distributed file service, which might be attempting concurrent access to the same file.

Several different cache-consistency strategies are used in existing systems. Two important examples are the NFS [13] and Sprite [7] file system protocols. (''Sprite'' is the name of an entire distributed operating system; we are concerned only with the Sprite file protocols, which we refer to as ''Sprite'' in this paper.)

NFS adheres to a stateless-server model and uses a probabilistic, stateless consistency scheme[1]. Sprite maintains server state, and uses an explicit consistency protocol. This protocol allows Sprite to guarantee the semantics of concurrent access by several clients to the same file; in addition, Sprite is alleged to provide better performance than NFS [7].

In NFS, cache consistency, performance, and crash vulnerability are inextricably linked together. NFS requires clients to write blocks immediately to the server; this write-back policy is necessary to maintain consistency, but performance is reduced. Through its consistency protocol, Sprite is able to separate performance from consistency, and thus improve both. The NFS write-through policy also limits the amount of data lost in a crash; Sprite allows clients to select this policy only when they need it. Although stateless servers have certain well-known advantages, they appear to be unable to simultaneously provide consistency and good performance.

In the experiment described in this paper, we transplanted the Sprite consistency protocol into the NFS file access protocol, hoping that some of Sprite's benefits would be transferred to NFS. This experiment also helps to isolate the effects of the cache consistency protocols from other differences between NFS and Sprite (for example, the different approaches to file name translation).

It proved to be relatively easy to modify the NFS implementation used in the Ultrix™ operating system to use the Sprite consistency mechanism. We call this system ''Spritely NFS[2].'' In section 3 we describe the specific changes to the NFS protocol, and in section 4 we describe our implementation.

[1]The official NFS protocol specification [16], while requiring a stateless server, says nothing about cache consistency. This specification provides insufficient guidance for producing a workable NFS implementation; the so-called ''reference port'' implementation is what actually defines correct behavior of NFS clients and servers. The reference port does imply a particular cache-consistency scheme.

[2] ''**sprite·ly** *adj.* [obsolete *spright* (sprite), alteration of *sprite*]: marked by a gay lightness and vivacity: SPIRITED **syn** see LIVELY'' [17].

Performance measurements, presented in section 5, are somewhat ambiguous. Depending on the benchmark, Spritely NFS either dramatically outperforms NFS, or performs slightly worse. Our expectation is that in actual use, Spritely NFS should perform moderately better than unmodified NFS. In any event, Spritely NFS guarantees that no two clients will have inconsistent cached copies of a file.

2. Approaches to cache consistency

We hoped to answer two questions in our experiment:

1. Are the performance advantages claimed for Sprite really the result of the cache-consistency mechanism, or are they attributable to differences in other features or to implementation quality?
2. Would adding Sprite-like consistency protocols to NFS improve NFS performance, and could this be done without significantly complicating the NFS implementation?

Implicit in these questions is the idea that the central difference between Sprite and NFS is their different approach to cache consistency. Indeed, the protocols are similar in many ways, particularly because both are meant to provide a nearly transparent emulation of the Unix® file system, using servers accessed by remote procedure call (RPC). In both Sprite and NFS, portions of a file may be cached in memory at the client (in contrast to the whole-file caching scheme used in Andrew [2] or Cedar [14]).

2.1. NFS consistency model

NFS follows a stateless-server model: the server retains no information about its clients between RPC requests, and all file data is written synchronously to disk. This simplifies the server implementation, avoids hard limits on the number of simultaneous clients, and makes server-crash recovery trivial.

Since the server has no record of which clients are currently using a file, it cannot itself guarantee cache consistency. An NFS client periodically checks with the server to see if a file has been modified[3]; if so, the client invalidates its cache for that file. The interval between checks is a compromise between performance (frequent checking loads the server and delays the client) and consistency (insufficiently frequent checking may mean that a client uses stale data from its cache). The check is also made each time the client opens a file.

Since one NFS client has no way of identifying other clients that may be concurrently accessing a file, all of its consistency checks must be made with the file server. Therefore, whenever a client modifies a file, it must immediately communicate the change back to the server. This "write-through" policy limits the potential inconsistency between the server's copy and the client's cache to a short period. It also limits the amount of damage caused by a crash; since an NFS server is required to write data to stable storage before returning from the remote procedure call, the amount of cached information that is vulnerable to loss during a crash is quite limited.

The use in NFS of write-through combined with periodic checks provides consistency as long as no client writes a file while another client has the file open. Because the notion of "open" is not present in the NFS protocol but is simply the state of a client with respect to a file, this is not a true guarantee, since there is no way to enforce it.

A write-through policy has two distinct performance disadvantages. First, write-through limits the performance benefits of client-side caching, since a server disk access is done for every write. A surprising number of Unix files have short lifetimes and are never shared by multiple clients [10], and thus need not be kept anywhere but in the cache of the client where they are created. NFS, unfortunately, cannot distinguish between shared and unshared files, and so must treat every file as if it were potentially shared. Both client and server waste effort performing unnecessary write-through operations.

The second disadvantage of a strict write-through policy is that it forces applications to run synchronously with the disk. While an application is waiting for the data to make its way over the network, through the server queues, and onto the disk, it is blocked. The application therefore takes longer to complete than it would if disk writes were performed asynchronously, as they are in the local Unix file system. Especially on single-user workstations, this time is wasted.

Actual NFS client implementations do not always write data synchronously. Instead, a block may be handed to a daemon process, which immediately writes it to the server; the original requesting process does not wait for the write to complete. This modification appears necessary for reasonable performance, but it does expose data to loss during a crash. In order to maintain consistency between opens of a file, an NFS client synchronously finishes all pending write-throughs when the file is closed.

2.2. Sprite consistency model

Sprite follows a stateful-server model. Unlike NFS, Sprite has explicit *open* and *close* operations. Since the *open* operation (in Sprite as in Unix) specifies if the application intends to write to the file, by tracking *open* and *close* operations the Sprite file server knows not only which clients are currently using a file, but whether any of them are potentially writers.

This is important because files are seldom "write-shared"; that is, seldom do two or more clients simultaneously have a file open that one of them is writing. More typically, either all the clients are doing read-only operations, or a single client "owns" the file while it is being modified. (We refer to these as the "read-only" and "single-writer" cases, or together as "non-write-shared.")

[3]The interval between probes in Ultrix varies between 3 and 60 seconds, depending on the recent history of the file.

Because a non-write-shared file can be cached at the clients without any danger of inconsistency, the Sprite server responds to each *open* request indicating if it is safe to cache the file. Clients can cache without the periodic consistency checks required in NFS. A single-writer client need not do write-throughs, and might never write to the server during the file's entire lifetime. (If reliability is more important than performance, an application can use explicit file-flushing operations to cause write-through.)

If a file is write-shared, none of the clients (writers *or* readers) are allowed to cache it. For writers, this reverts to the write-through policy of NFS, and provides the same single-copy consistency between a writer and the server. For readers, this is stricter than the NFS mechanism; each read operation goes directly to the server. Readers are guaranteed consistency with writers, provided that some other mechanism (such as file locking) serializes the reads and writes. NFS cannot feasibly use such a strict non-caching policy, since NFS cannot distinguish between the infrequent write-sharing situation when it is beneficial, and the normal non-write-shared case when it is wasteful.

Of course, when multiple clients open a file, they do not all issue their *opens* simultaneously. When a file that is open read-only is subsequently opened for write, the Sprite file server must notify not only the newly-arrived writer, but also the existing readers, that the file is no longer cachable. (Similarly, when a file is opened first by a single writer and then by another client, the first writer must be told to stop caching its copy and to return all the dirty pages to the server.) To do this notification, the Sprite server makes asynchronous calls to the client, or ''callbacks.'' Callbacks are the other major difference between the NFS and Sprite protocols (in addition to the explicit *open* and *close* operations).

There are other design differences between Sprite and NFS that are not further considered in this paper, although they do significantly effect performance. For example, Sprite servers can translate entire (multiple-component) file pathnames in one operation, whereas NFS servers translate pathnames one component at a time. Sprite and NFS also use different RPC protocols.

2.3. Potential advantages of Sprite

The Sprite consistency mechanism, unlike the periodic checks used in NFS, *guarantees* consistency between clients accessing the same file. Thus, in some sense Sprite is more ''correct'' than NFS. We do not know how important this is in practice: since application writers know that NFS does not provide true consistency, and especially because write-sharing is infrequent in any case, the lack of consistency in NFS may not be significant. (A more frequent case is ''sequential write-sharing,'' where the writer closes a file before the reader opens it; NFS provides consistency in this case.) On the other hand, the weakness of NFS consistency may be responsible for the lack of shared-database applications.

Sprite should also provide better performance. Most important is its use of write-back instead of write-through. This improves performance in two cases:

1. An application that alternates computation with disk output (such as the compilation of several modules) can do both in parallel, since Sprite allows the client's writebacks to proceed asynchronously even across file closes.
2. An application that generates short-lived files (such as a compiler with its intermediate files, or a sort program) need not ever pay the cost of writing them to disk. The client can delay write-back long enough that the file may be deleted before the file is ever written to the server. This becomes more significant as memory chip sizes, and consequently client file system cache sizes, increase.

These two cases are quite common for typical Unix workloads, especially on diskless workstations, and the performance improvements can be dramatic. Reducing the number of server writes improves response time, since writes are always synchronous with the disk at the server, unlike reads which often hit in the server cache. Reducing server writes can also improve the read hit rate of the server cache, since ''useful'' cached data is less likely to be replaced by the data from ''useless'' writes.

Sprite avoids the cost of periodic consistency probes, but instead must do explicit *open* and *close* operations. In Sprite (but not Spritely NFS) the *open* operation is ''piggybacked'' on the file name translation, thus eliminating one RPC call. Even so, in the case where a file is opened, read quickly, and then closed, Sprite would require one more RPC operation than NFS (because NFS would not need to do any consistency probes in such a brief interaction).

If the application mix is right, use of the Sprite consistency mechanism should improve performance over NFS by reducing client-to-server traffic and server disk I/O. The latter is especially important because disk access times are not improving nearly as fast as processor and communication speeds. This in turn should improve client response time, and also increase the number of clients that can actively use a single server (and thus that can actively share a single file system).

Because an NFS server keeps no per-client or per-open-file state, in theory it could handle an arbitrary number of clients or open files. A Sprite server cannot serve an unbounded number of clients or files, since it keeps information about each recently-active file. That comparison may be illusory: while the NFS server may be able to ''handle'' an arbitrary number of clients, the Sprite server should be able to provide acceptable performance to a larger number of simultaneously active clients.

2.4. Implications for crash recovery

Because NFS servers are stateless, server crash recovery in NFS is trivial: the server simply restarts. Client crash recovery is also fairly simple, since all client data is written immediately (as soon as possible) to disk, and synchronously on close[4]. Only crashes that occur between the creation of data by an application (for example, keystrokes to a text editor) and the completion of a file-write RPC cause data loss; this is actually better than the local file system reliability in Unix, where a disk write may be delayed for as much as 30 seconds.

Sprite provides roughly the same protection against client crashes as does a local Unix file system. (An application can always do explicit file flushes to provide crash-resistance, but few existing Unix programs are written this carefully. As in Unix, the write-delay period may be adjusted to reduce the crash-vulnerability window.) Sprite server crash recovery is more complex than in NFS, since the server must reconstruct the state it maintains about which files are open by which clients (or else the clients could become inconsistent). A server-crash recovery mechanism has been implemented for Sprite [18]; it relies on two of Sprite's properties:

1. The clients together "know" who is caching the file, and the server can reconstruct its state from the clients.
2. The consistency state of the file cannot change while the server is down, or until the server is willing to allow it to change.

Most of the complexity in the recovery mechanism comes in detecting crashes and reboots, rather than in rebuilding state. This is done by tracking the passage of RPC packets, and using periodic "keepalive" packets, to detect when a client or server has crashed or rebooted; the same mechanism also suffices to detect network partitions. There is some cost to tracking RPC packets, but a reliable crash and reboot detection mechanism is of course useful for other purposes besides recovering file server state.

2.5. Related work

A cache-consistency mechanism roughly intermediate between that of NFS and Sprite has been implemented for the System V Remote File Sharing (RFS) system [1]. As in NFS, clients write-through to the server, so the only possible inconsistency is between the server and readers. RFS is not stateless; clients send *open* and *close* messages to the server, so the server is able to send "invalidate" messages back to clients when their caches must be disabled. Unlike Sprite, RFS waits until writes actually occur before invalidating client caches. As in both Sprite and NFS, version numbers are used to maintain client cache consistency when a file is reopened after being closed. RFS provides the same consistency guarantees as Sprite, but because RFS uses the same write policy as NFS, its performance should be closer to that of NFS.

Both Sprite and RFS use entire files as the unit for consistency. Kent [4] describes a system that maintains consistency on individual file blocks; before a client writes a block, it must acquire ownership of that block. Other clients invalidate cached copies of that block, and only one client at a time can own a block. This system required special hardware to implement the consistency protocol with sufficient performance.

The dogma of statelessness associated with NFS has been broached before. Juszczak [3] shows that because the individual NFS operations are not really idempotent, certain kinds of communication failure can result in incorrect behavior. A similar observation is made in [2]. By adding a small amount of state to the NFS server, he managed to resolve this problem, and also to improve the performance of highly-loaded servers.

Many other file system designs include explicit consistency protocols. These include Andrew [2], Cedar [14], Apollo [6], and Locus [11]. A comparison between Sprite and these systems may be found in [7].

3. Modifications to the NFS protocol

In this section we sketch the modifications to the NFS protocol necessary to support the Sprite consistency protocols (see [15] for a detailed description). This is the "Spritely NFS" (SNFS) *protocol* most of the complexity in SNFS is in the *implementation*, described in section 4.

3.1. New client-to-server calls

In unmodified NFS [16], all RPC calls are initiated by the client. We added two new calls, *open* and *close*, to the client's repertoire.

The *open* RPC call takes a "file handle" (an identifier returned by the lookup operation) and a flag indicating if the client intends to write the file. The server returns a `cacheEnabled` flag to tell the client whether it is allowed to cache data for this file. The server keeps a version number for each file, which increases every time the file is opened for writing; the open call returns both the latest version number and the previous version number. A client's cache is valid if the latest version number matches the version of the cached copy. If the client is opening the file for write, its cache is also valid if it matches the previous version number; this is because the change in version number has resulted from the current open-for-write. The server also returns the same `attributes` record that would have been returned from a *getattr* (get file attributes) procedure, obviating the *getattr* call that NFS must make when a file is opened.

The *close* procedure tells the server that the client is no longer using the specified file handle. The client passes the same `writeMode` flag that it provided for the corresponding open operation; it must be supplied since *open* could have been called several times, with different modes, on a single file handle.

[4]Actually, the reference port of NFS delays writes that do not extend to the end of a block, as a means of optimizing improperly-buffered sequential writes.

3.2. Server-to-client calls

Whenever an SNFS server needs to notify a client that a file must no longer be cached, it issues a callback operation. This RPC call goes from server to client, so the client must provide RPC service for this request. The callback operation identifies the file in question through several parameters that allow the client to locate its data structures for the file. Two flags specify what actions to take: one indicates that any dirty blocks in the client's cache should be written back to the server, and the other that any cached blocks should be invalidated and further caching disabled.

If the callback involves a write-back, the client should not return from the callback RPC until all the dirty blocks have been written back to the server. This has two implications:

1. Because the write operations generated by the client in response to the callback go to the server that is waiting for the termination of the callback, an SNFS server must be multi-threaded to avoid deadlock. If there are *N* threads, only *N-1* may be doing callbacks simultaneously, so that at least one thread can service the write-backs.
2. Since the server makes a callback while servicing an open operation from another client, it cannot wait forever for the callback (since the client doing the open will time out). Usually the callback, together with any required write-backs, should finish long before the RPC times out, but this is not guaranteed; the network might be slow, the server might be overloaded, or there might be many dirty blocks. We believe that this is not a serious problem; the callback operation can safely be retried, so when the client doing the open operation times out and retries, no harm is done. (Care must be taken to avoid delayed duplicate RPC packets.)

If the client "serving" the callback is down, the SNFS can honor the new open operation, but it should inform the new client that the file may be in an inconsistent state. (This is hard to do in the Unix model.) If the "dead" client comes back to life after this point, it must be prevented from making further use of the file until it obtains a new file handle and reopens the file.

4. Implementation

We implemented a prototype of Spritely NFS by modifying the existing NFS implementation in Ultrix version 2.2. By changing the names of entry points and global variables, we made it possible to have both SNFS and unmodified NFS in the same kernel, which in turn made it easy to compare the performance of the two protocols. With the exception of a few utility programs, all the changes are confined to the kernel; for user code, there is no visible difference between NFS and SNFS.

4.1. Layering

In Ultrix, the "generic file system" (GFS) [12] (see also [5]) provides a level of indirection to separate the filesystem-generic code from the filesystem-specific code. GFS implements the *gnode* abstract data type, which is similar to the traditional Unix in-memory *inode* data structure, but which supports multiple file systems (such as NFS and local disks). GFS manages the file system block buffer cache, and expects the underlying file systems to export a set of methods for reading and writing file blocks. We needed only one minor modification to GFS to support SNFS (see [15] for implementation details not discussed here).

On the server side, the NFS (and SNFS) service code simply translates RPC requests into GFS operations on the appropriate file system, normally the standard Unix local file system.

4.2. Client changes

The *gnode* data structure provides space for filesystem-specific data, some of which is already used by the NFS client code. We added several new fields, including flag bits such as "caching enabled", the file version number, and authorization information used when doing a delayed write. No additional state tables are needed at the client.

GFS invokes the *open* and *close* entry points for all file system types, including NFS clients, when a file is opened or closed. The new RPC calls in SNFS are dispatched from these routines. (Many of the *close* calls could be avoided; see section 6.2.)

4.2.1. Cache strategy

Two kinds of information are cached on the client: file data blocks and file attributes. The file data blocks are cached in the GFS buffer pool; each block is marked with the appropriate file ID. The file attributes are stored in the *gnode*.

Ultrix NFS refreshes the attributes cache based on its age; an adaptive mechanism is used which allows longer residence for files that have not been recently modified. In SNFS, the attributes cache needs no refreshing if the file is read-shared (cachable). If the file is write-shared (not cachable), SNFS guarantees consistency by always fetching attributes from the server, instead of caching them.

NFS maintains consistency for cached data by checking the file modification timestamp, and invalidating the cache if the timestamp changes. In SNFS, the explicit consistency protocol maintains the cachability flag for the file; if the file is not cachable, its blocks are never entered into the cache. Also, the standard Unix read-ahead is disabled in SNFS for non-cachable files, since the extra blocks cannot be cached.

4.2.2. Callback service

In unmodified NFS, all RPC calls are initiated by the client. In SNFS, the server initiates callback RPCs, so the client must be able to service RPC requests. We simply use the existing NFS server code.

The callback RPC is implemented as part of the SNFS server code, but conceptually it is part of the SNFS client.

The information in the callback is used to locate the *gnode* for the specified file, and the specific action is performed. Cache invalidation is done locally to the client; if the server requests write-back, the client uses the usual SNFS RPC calls to write blocks back to the server.

4.2.3. Delayed write policy

Traditional Unix policy is to delay file data writes to the local-disk file system, unless a user process explicitly flushes them. Blocks are written back to disk when the space is needed for other files. To bound the amount of damage caused by a crash, all delayed-write blocks are written to disk periodically (every 30 seconds, by the *sync* system call from `/etc/update`). In the Sprite file system, dirty blocks are written back to the server when they reach 30 seconds in age; this is somewhat less conservative than the traditional policy.

The NFS consistency mechanism prevents the accumulation of delayed-writes. SNFS, on the other hand, uses the normal GFS delayed-write mechanism, so (mostly by default) it follows the traditional Unix policy.

Since it is relatively common for Unix applications to create a temporary file and then delete it after a few seconds, Sprite and SNFS take advantage of this behavior by ''cancelling'' delayed writes when a file is deleted. NFS cannot do this, since it synchronously writes back on close.

4.3. Server state design

4.3.1. Server state table

An SNFS server, unlike an NFS server, must retain state about files between RPC calls. In our implementation, the SNFS server maintains a state table, organized as a hash table, with entries for each open file and for each closed file whose last writer may still have cached blocks.

To avoid running out of kernel memory, we limit the number of entries in this table. This limits the number of simultaneously open files per server, a limit that is not imposed by unmodified NFS (but each entry requires only 68 bytes, so the limit can be liberal). When entries run low, those recording closed files may be reclaimed by sending callbacks to the corresponding clients.

Most of the code added to support SNFS is in the state table manager module. It has entry points to initialize the server state data structures, and to perform the state transitions necessary on file *open* and *close* operations.

Our only modification to the original NFS server code was to add the two new RPC services functions. The *open* operation is similar to the existing *getattr* operation, but it calls the state table manager to record information about the new open, potentially resulting in a callback. The *close* operation does nothing but notify the state table manager.

4.3.2. State table entries

Each entry in the state table contains the file handle for the corresponding file; this is the lookup key. It also contains the file's current version number, its current state (such as read-only or write-shared), and a list of ''client'' information blocks for each client host that has the file open, or that might have dirty blocks in its cache if the file is closed.

A client information block contains the network address of the client host; this is used as an identifier and also to address the callback RPCs. A client block also contains counts of the number of readers and writers for this file at this client (more than one process there may have the file open) and additional information used in the callback RPC to help the client identify the file.

4.3.3. Version number generation

The server assigns a version number to each file; the version number must increase each time the file is opened for writing. This allows clients to determine if cached blocks are still valid when a file is reopened. Ideally, the version number would be associated with each file on stable storage (as is done in Sprite), but since we did not want to modify the underlying Unix local file system to store additional information, we chose to use a global counter to generate version numbers. This solution is suitable only for experimental use, as it poses several obvious problems.

4.3.4. State transitions

Each file may be in one of several states. There is some freedom in the choice of state assignments; in retrospect, the one we chose would have to be changed to support ''delayed close'' (see section 6.2) without deadlocking. In our implementation of SNFS, the states are:

CLOSED	File not open by any client.
CLOSED_DIRTY	File not open, but the last writer may still have dirty blocks.
ONE_READER	File open read-only by one client.
ONE_RDRDIRTY	File open read-only by one client, which may have dirty blocks cached from a previous open.
MULT_READERS	File open read-only by two or more clients.
ONE_WRITER	File open read-write by one client.
WRITE_SHARED	File open by two or more clients, including at least one writer.

Table 4-1 shows the possible state transitions. Note that no transition occurs (and thus none is shown) if a client that already has a file open for read-only issues another read-only *open* for that file, or if a client that has a file open for read-write issues another *open* of any sort for that file.

FROM STATE	TO STATE	WHEN	CACHING	CALLBACK
CLOSED	ONE_READER	Open for read	Enabled	None
CLOSED	ONE_WRITER	Open for write	Enabled	None
CLOSED_DIRTY	ONE_RDRDIRTY	Open for read by last writer	Enabled	None
CLOSED_DIRTY	ONE_READER	Open for read not by last writer	Enabled	Write-back
CLOSED_DIRTY	ONE_WRITER	Open for write by last writer	Enabled	None
CLOSED_DIRTY	ONE_WRITER	Open for write not by last writer	Enabled	Write-back and invalidate
ONE_READER	MULT_READERS	Open for read by different client	Enabled	None
ONE_RDRDIRTY	MULT_READERS	Open for read not by last writer	Enabled	Write-back
ONE_READER or ONE_RDRDIRTY	ONE_WRITER	Open for write by same client	Enabled	None
ONE_READER	WRITE_SHARED	Open for write by different client	Disabled	Invalidate
MULT_READERS	WRITE_SHARED	Open for write	Disabled	Invalidate
ONE_RDRDIRTY	WRITE_SHARED	Open for write by different client	Disabled	Write-back and invalidate
MULT_READERS	MULT_READERS	Open for read	Enabled	None
ONE_WRITER	WRITE_SHARED	Open for read or write by different client	Disabled	Write-back and invalidate
MULT_READERS	ONE_READER	Close by last but one reader	Not affected	None
ONE_READER or WRITE_SHARED	CLOSED	Final close	Not affected	None
ONE_RDRDIRTY	CLOSED_DIRTY	Final close	Not affected	None
ONE_WRITER	CLOSED_DIRTY	Final close	Not affected	None, this client recorded as last writer
ONE_WRITER	ONE_RDRDIRTY	Final close for write, client still reading	Not affected	None, this client recorded as last writer

Table 4-1: SNFS server state transitions

4.4. Crash Recovery

We have not yet implemented a crash recovery protocol for Spritely NFS. Such a protocol would involve changes to both the client and server implementations, and would reduce performance to some extent. Note that the published measurements of Sprite [7] were also made without a recovery protocol.

4.5. Code size

A crude measure of the complexity of the modifications we made is the change in source code size. The unmodified NFS code we started with consisted of 9200 lines of commented C source code in 15 files. The SNFS version consists of 11150 lines in 16 files, most of the increase coming from the SNFS server state manager. We believe that an implementation supporting both NFS and SNFS protocols would be only a few hundred lines longer than our SNFS code, although crash recovery code would be substantially larger.

Run-time data space requirements vary depending upon the limit imposed on the number of open files; for example, up to 1000 simultaneously open files can be accommodated with about 70 kbytes of data space.

5. Performance

In this section, we look at the performance differences between NFS and SNFS. We focus on the most common case, where there is no concurrent sharing of a file between two or more client hosts. In the write-shared case, SNFS disables the client cache and so performs much worse than NFS — but much more correctly.

5.1. Factors affecting performance

The performance differences between NFS and SNFS are the result of variation in several factors, which depend on the application mix:

- The parallelism available with delayed write instead of write-through.
- The writes averted when temporary files are deleted before being written back.
- The number of RPC calls required over the active lifetime of a file.

We believe that the computational costs of the SNFS implementation are not significantly different from those of NFS.

For example, SNFS gains most from increased parallelism when only one job is running on the client host, and it can alternate computation with write I/O (such as a compiler). File copying can also benefit as long as the cache does not fill with dirty blocks, because the writes are often postponed so as to overlap with other tasks. Less such I/O parallelism is available if many applications are running in parallel on the client.

Similarly, SNFS gains by avoiding writes if the application is generating a significant volume of temporary files (and if these files fit easily into the client cache).

The relative number of RPC operations depends on the application. For example, a file that is read only once for a brief period (such as a source module) differs from a file that is read over the course of several seconds (some text editors do this, for example). In the "read-quickly" case, NFS will require one fewer RPC than SNFS, since SNFS requires the additional *close* operation (the SNFS *open* operation is equivalent to the *getattr* operation done at file-open time by NFS). In the "read-slowly" case, SNFS may break even or better, since NFS must do consistency probes every few seconds.

Frequently, the NFS model wins because most applications follow the "read-quickly" pattern. As we point out in section 6.2, however, a minor modification to our implementation of SNFS could perform significantly better than NFS in the case where a file, such as a popular header file, is read repeatedly during the course of some seconds. This pattern is actually quite common.

In addition to effects of the application mix, the relative performance of SNFS and NFS depends on system parameters including the file cache size, RPC speed, and disk access time. As the client's file cache size increases, the relative benefit of clever cache-management protocols increases as well. Also, when the gap between processor speeds and disk access time widens (as it appears to be doing), cache-management efficiency becomes more important. Finally, since NFS and SNFS differ somewhat in the number of RPC calls used, an increase in RPC speed (relative to processor speed) reduces the relative performance difference.

5.2. Andrew benchmark measurements and analysis

Our SNFS implementation was originally developed for Ultrix running on a MicroVAX-II™ with a relatively small memory. Because we were interested in the effects of large caches, we ported the code to the experimental Titan workstation. Titans are RISC processors running about 12-15 times as fast as a VAX-11/780, and supporting up to 128 Mbytes of main memory [8]. Identical machines were used for client and server, and the RA81 and RA82 disks used are moderately high performance drives. The operating system running on the Titan is not exactly Ultrix, but the NFS and other file system code is taken directly from Ultrix, with only a few lines changed because of architectural differences. All our measurements were made on Titans.

It is relatively easy to benchmark the individual cases where one might expect SNFS performance to differ from NFS performance. It is harder to measure an aggregate difference, since the weighting for the individual differences depends so much on the application mix. We chose to concentrate on the Andrew benchmark suite [2], since it covers many of the individual cases and does give some idea of the aggregate performance. The Andrew benchmark spends a significant amount of time doing compilation; since the cost of compilation depends upon the target architecture, it is not possible to compare our figures directly to previously published results from the Andrew benchmark[5]. We also benchmarked an external sort application, since this emphasizes the differential performance on temporary files; see section 5.3.

The Andrew benchmark consists of 5 phases, applied to a tree of directories and files; the following description is taken from [2]:

MakeDir	Constructs a target subtree that is identical in structure to the source subtree.
Copy	Copies every file from the source subtree to the target subtree.
ScanDir	Recursively traverses the target subtree and examines the status of every file in it; does not actually read the contents of any non-directory file.
ReadAll	Scans every byte of every file in the target subtree once.
Make	Compiles and links all the files in the target subtree.

Different phases highlight different differences between SNFS and NFS. The *Copy* phase favors SNFS, since the delayed-write policy allows more parallelism between the read and write I/O streams. The *ScanDir* and *ReadAll* phases favor NFS, since SNFS has about one additional RPC to do for each file. The *Make* phase favors SNFS because it allows parallelism between file writing and either file reading or computation.

Because the delayed-write policy of SNFS postpones some operations until after the completion of the benchmark, we ran the SNFS benchmarks several times in a row (rather than interleaving them with NFS benchmark runs) so that NFS would not be charged for writes incurred by SNFS.

We ran the benchmark in three configurations: one with all files on the local disk, one with just the data files remotely mounted but temporary files kept locally, and the last with both data and temporary files remotely mounted. The latter configuration should favor SNFS for the *Make* phase, since it allows the "delete-before-writeback" optimization to take effect. In all configurations, the

[5]We used a slightly modified version of the original Andrew benchmark, due to John Ousterhout [9], that does produce comparable numbers. This is done by using a portable compiler and loader that produce code for a fixed target architecture, not for the architecture being tested. We hope that future benchmarking will be based on this portable version.

''compiler'' programs were on the same file system as the data, and other Unix utility programs were on the local disk.

The results are shown in table 5-1. Each number shown is an average over 10 trials. Measurement accuracy is no better than a second or two, so slight variations are meaningless.

During our experiments, neither the client nor server machine were used for any other jobs (although some housekeeping tasks occasionally run in the background). Both machines had large file buffer caches (about 16M bytes on the client and 3.5M bytes on the server), large enough that no data was ever removed from the caches due to replacement. This simplifies analysis but does favor SNFS, which is better able to make use of a large cache than NFS.

The results shown in table 5-1 confirm our expectations. SNFS performs about 25% better on the *Copy* phase, and 20% to 30% better on the *Make* phase (depending on whether `/tmp` is local or remote). NFS performs about 5% better on the *ScanDir* and *ReadAll* phases. SNFS completes the entire benchmark 15% to 20% faster than NFS, because the complete benchmark places most weight on the *Make* phase.

Elapsed time in seconds					
Phase	Local	NFS, /tmp local	SNFS, /tmp local	NFS, /tmp remote	SNFS, /tmp remote
MakeDir	4.7	4.6	4.6	4.5	4.3
Copy	24	48	35	48	37
ScanDir	43	52	55	52	55
ReadAll	37	50	53	51	53
Make	215	303	237	377	266
Total	322	457	384	532	415

Table 5-1: Results of Andrew benchmark

Table 5-2 shows RPC operation counts for a typical trial with each of the NFS and SNFS configurations; there may be insignificant inaccuracies in the counts. With `/tmp` on a local disk, SNFS requires slightly (2%) more RPC operations, but since SNFS substitutes *open* and *close* operations for the more expensive *read* and *write* operations, it comes out ahead in total cost. With `/tmp` remotely mounted, SNFS requires 6% fewer total operations, and 42% fewer data transfer operations.

Remote Procedure Calls				
Call	NFS, /tmp local	SNFS, /tmp local	NFS, /tmp remote	SNFS, /tmp remote
open		700		778
close		700		776
getattr	757	225	933	311
setattr	22	22	22	22
read	1130	699	1961	1033
write	868	590	1425	921
lookup	3345	3345	3543	3543
other	273	273	376	376
Total	6395	6554	8260	7760

Table 5-2: RPC calls for Andrew benchmark

Several entries in table 5-2 deserve explanation. When `/tmp` is mounted locally, one might expect both protocols to issue the same number of *write* RPC calls. Because the Ultrix NFS implementation delays partial-block writes, it is more sensitive than SNFS to the ''natural'' file system block size used at the server. During our tests, we used a 4k byte block; NFS might have performed slightly better had we used an 8k byte block size.

We also found that NFS issues far more *read* RPC calls. In trying to explain why this is so, we discovered that it is not the fault of the NFS protocol. Rather, our version of the NFS code invalidates the client data cache when a file is closed. In many instances the client first writes a file, closes it, and then reopens and reads it, and this bug prevents the client from using its cached copy. Our NFS implementation is based on a version of the reference port that is several years old; more recent implementations of NFS have fixed this bug, but we were unable to measurement their performance on comparable hardware.

Finally, we note that roughly half of the RPC calls are file name lookups (SNFS and NFS use the same protocol for this). Clearly, any mechanism that reduced the number of lookups would improve performance; we suspect that applying the Sprite consistency protocols to a cache of directory entries might be a good approach[6].

[6]Recent versions of NFS also do more extensive caching of name translations.

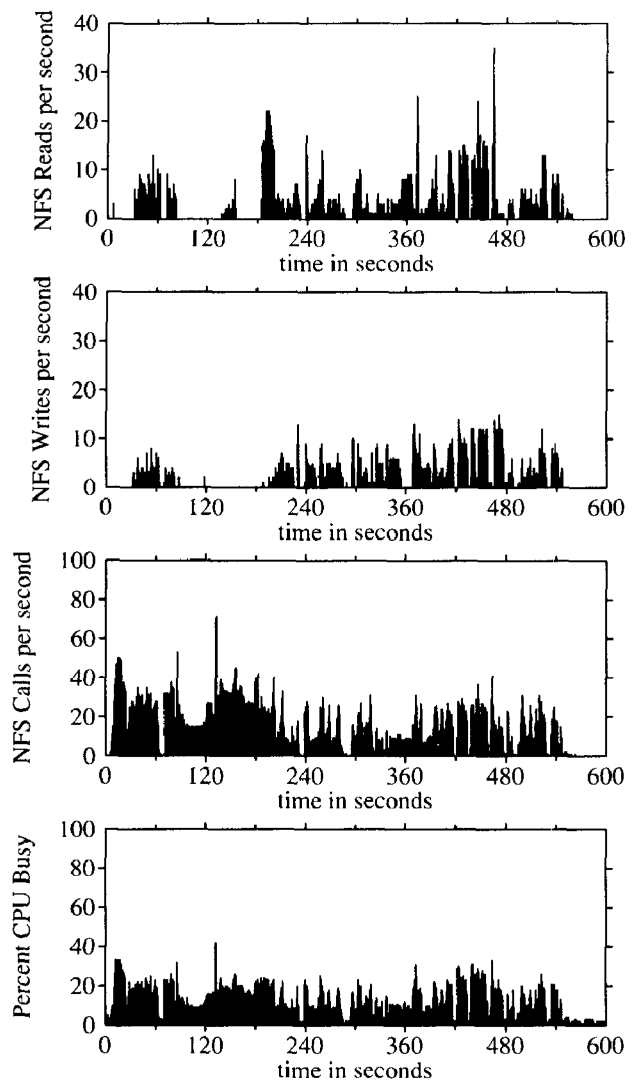

Figure 5-1: Server utilization and call rates for NFS

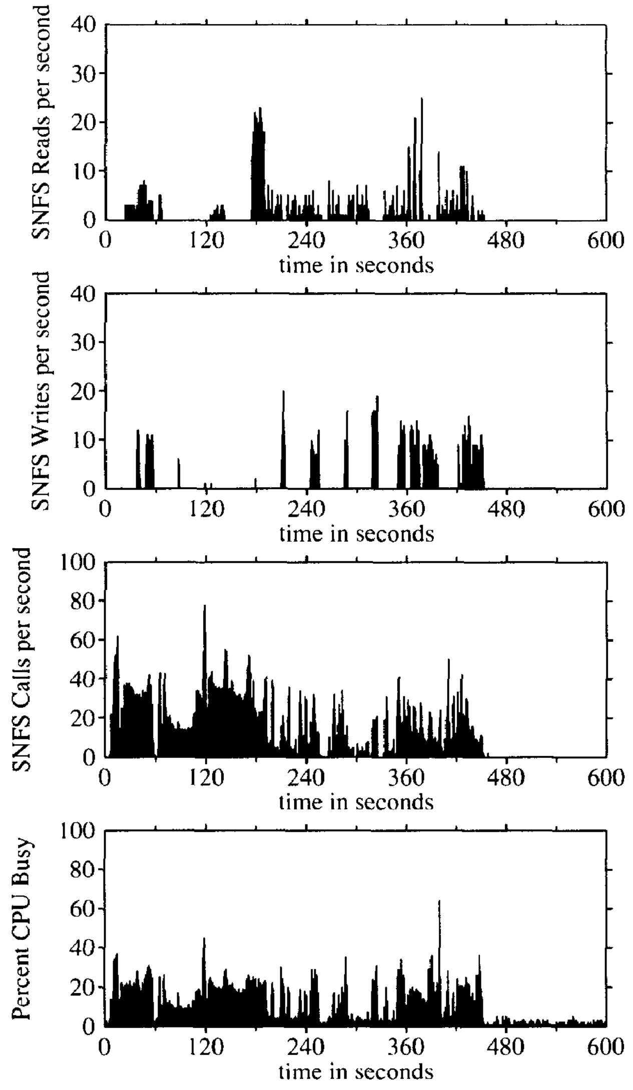

Figure 5-2: Server utilization and call rates for SNFS

We were also interested in the effect of file system protocol on "server utilization," the CPU load placed on the server for a given application. Measurements of the Sprite operating system suggest that the Sprite file system can support about four times as many clients as can a Unix system with NFS running on identical hardware [7]. We measured the server CPU load (roughly, the percentage of time not spent in the "idle" state) while running the Andrew benchmark for NFS and SNFS; in both cases, `/tmp` was remotely mounted, effectively simulating the load of a diskless workstation. We also measured the rate of RPC calls, as well as individual rates for *read* and *write* calls. Graphs of the server load and call rates are shown in figure 5-1 for the NFS benchmark, and in figure 5-2 for the SNFS benchmark. All the graphs in one figure are for the same run, so one can see how the rates are correlated in time.

The load varied considerably over the course of the benchmark, and was strongly correlated with the aggregate rate of RPC calls; it was *not* correlated with the rate of *read* or *write* calls. Since SNFS, even when `/tmp` is remotely mounted, requires only slightly fewer operations than NFS, the integral of CPU load over time was only slightly lower for SNFS. In fact, since the SNFS benchmark completes significantly faster, the average server load during the benchmark is slightly higher than for NFS; it also appears to be slightly burstier.

We believe that the advantage, in server CPU utilization, of Sprite over NFS is probably the result of a more efficient RPC protocol and perhaps a more efficient file name translation mechanism. We have no evidence to show that the SNFS cache consistency protocol itself, in isolation from the write policy, leads to significantly different server CPU utilization on the Andrew benchmark. On the other hand, the difference in *write* operation rates (see table 5-2) implies that the server disk utilization with SNFS is 30% to 35% lower.

5.3. Sort benchmark measurements and analysis

The Andrew benchmark suggested that the most significant difference between NFS and SNFS was their performance on temporary files. (This is most important for diskless clients.) We explored this case by benchmarking the Unix *sort* program, which does an external sort and so makes heavy use of temporary files. (This benchmark also emphasizes the performance degradation caused by the inability of our NFS client implementation to retain cached data after closing a file.)

We measured the performance of the sort program with its temporary files (kept on `/usr/tmp`) on local disk, remote-mounted via NFS, and via SNFS. Table 5-3 shows the resulting elapsed times for input files of three different sizes; the important parameter is the amount of temporary storage used, which grows faster than the input file.

SNFS dramatically outperforms NFS on this benchmark, completing approximately twice as fast. In all three cases the client CPU utilization is higher for SNFS; in other words, I/O latency is the bottleneck. Table 5-4 shows that SNFS does far fewer *read* RPC calls than does NFS, indicating that some of the difference is attributable to the bug in our NFS implementation, rather than the NFS protocol itself. We believe that this accounts for less than a quarter of the performance difference, the rest attributable to the synchronous writeback-on-close required in NFS.

File size	Temp storage	local /usr/tmp	NFS /usr/tmp	SNFS /usr/tmp
281 k	304 k	4 sec	8 sec	4 sec
1408 k	2170 k	33 sec	105 sec	48 sec
2816 k	7764 k	74 sec	234 sec	127 sec

Table 5-3: Results of Sort benchmark

Remote Procedure Calls				
Call	1408 kbytes, NFS	1408 kbytes, SNFS	2816 kbytes, NFS	2816 kbytes, SNFS
open		35		68
close		35		67
getattr	93	37	150	70
read	681	33	1340	67
write	729	706	1452	1441
other	108	107	203	207
Total	1611	953	3145	1920

Table 5-4: RPC calls for Sort benchmark

We ran a simple benchmark on a recent NFS implementation (SunOS Release 4.0.3 running on a Sun-3) to highlight the penalty for invalidating the client cache when closing a temporary file. This benchmark writes a large file, closes it, and then opens and reads either the same file, or a different file of the same size. We caused the client cache to be invalidated between trials. There was no significant difference in elapsed times, indicating that the (elapsed-time) cost of a read missing the client cache is negligible compared to the cost of writing through.

Unlike the Andrew benchmark, on the sort benchmark the total server CPU utilization is about 40% lower for SNFS, probably because SNFS does about 40% fewer RPC calls. This is a significant improvement, but might disappear with a more careful NFS client.

5.4. Avoiding file writes for temporary files

A delayed write policy means that data written to short-lived temporary files may never need to be sent to the server. The sort benchmark runs long enough that the periodic write-back done by the Unix `/etc/update` process is likely to cause significant traffic even though few of the temporaries actually reach the age of 30 seconds.

To emphasize the benefits of delaying writes of temporary files, we ran the sort benchmark with the `/etc/update` process disabled. The results, shown in table 5-5, show that for files whose lifetime is short enough, SNFS matches or beats local-disk performance (even though data blocks are not written, the local-disk file system still writes out structural information). NFS performance is unchanged, within the limits of measurement error. Table 5-6 shows that SNFS, in this situation, is doing almost no *write* RPC operations.

File size	Temp storage	local /usr/tmp	NFS /usr/tmp	SNFS /usr/tmp
1408 k	2170 k	32 sec	97 sec	29 sec
2816 k	7764 k	69 sec	246 sec	69 sec

Table 5-5: Sort benchmark, infinite write-delay

Remote Procedure Calls				
Version	`update`?	Reads	Writes	Others
NFS	Yes	1340	1452	353
NFS	No	1227	1451	368
SNFS	Yes	67	1441	412
SNFS	No	65	33	407

Table 5-6: RPC calls for Sort benchmark, 2816 kbyte input file, with infinite write-delay

6. Future work

In this section we touch on several issues we have not yet addressed in our implementation. (We have already touched on the issue of crash recovery in section 2.4.)

6.1. Coexistence of NFS and SNFS

SNFS coexists quite easily with unmodified NFS. A single client host can remote-mount file systems using either protocol, and a single server host can provide access to separate file systems using either protocol. A hybrid server could distinguish between SNFS and NFS clients because SNFS clients always perform open operations before other file operations. A hybrid client could distinguish between SNFS and NFS servers, since the latter will reject an open operation. Thus, the SNFS clients and servers will discover each other, and other combinations will simply revert to the standard NFS protocol.

It is trickier to support simultaneous access via both NFS and SNFS to the same file system, since the NFS clients cannot participate in the SNFS consistency protocol. One approach is to treat any NFS access to a file already open under SNFS as implying an SNFS open operation. The server also has to keep, for a period no less than the longest reasonable NFS attributes-probe interval, a record of all other files accessed via NFS. By using this information, the server can manage the caches of SNFS clients so as to guarantee their consistency, and still provide ''normal'' NFS consistency to the NFS clients. (See [15] for more details.)

6.2. Delaying the SNFS close operation

Our SNFS implementation sends an open operation to the server every time a process opens a file. This is not necessary; since most files are reopened soon after they are closed, we could avoid a lot of network traffic if the SNFS clients delayed close operations in anticipation of a subsequent open operation. The client would keep a flag in the *gnode* structure indicating that a ''closed'' file has not yet been reported to the server; this would allow it to realize that a subsequent open operation can be performed locally.

Delayed-close may create situations where the server perceives write-sharing to be taking place, when in fact it is not. If a client with a delayed-close file receives a callback for that file, the appropriate response is to close the file so that it can be cached by the new client host. Delayed-close will also cause the server's state table to fill up with apparently open files. It may be necessary to create a new callback mode that asks a client to relinquish a closed file; the server would perform these as necessary to attempt to reclaim state table entries that have not been used recently. Clients could also spontaneously issue close operations for files that have not been re-opened after a few minutes.

7. Summary and Conclusions

Our experiments have convinced us that the Sprite approach to consistency is superior to that used in NFS. NFS cannot provide complete consistency with acceptable performance. Even with the weak consistency provided by most NFS implementations, performance is probably worse than that provided by the Sprite consistency protocol. Sprite's performance advantage over NFS comes mostly from its delayed write-back policy, not directly from the explicit cache consistency protocol, but without such a protocol, delayed write-back is too dangerous.

We found that adding the Sprite consistency protocol to NFS was possible without major disruption of the NFS implementation, and required only a few programmer-months. In order to completely refute the dogma of statelessness, we would have to demonstrate that SNFS has the same fault-tolerance as NFS; this would require implementation of a recovery protocol.

We did not find that SNFS outperformed NFS as much as Sprite itself outperformed NFS [7]. One reason may be that the NFS we used has been adjusted to place performance ahead of consistency; perhaps this is the right choice. A more intriguing question is whether the high rate of file lookup calls, as we detailed in table 5-2, swamps other file system performance differences. NFS and SNFS use the same lookup mechanism; Sprite uses an entirely different approach, which might account for its advantage, and might profitably be applied to the NFS protocols.

Caching in file systems is becoming more crucial as processor speeds and memory sizes improve faster than disk access times. We cannot afford to use inadequate cache mechanisms simply because the good ones seem harder to implement.

8. Acknowledgements

Richard Swan prompted our interest in integrating Sprite concepts into a quotidian operating system. John Ousterhout, in his post-sabbatical semi-residence at our laboratory, helped us understand Sprite and kept us honest. Richard Hyde and Chris Kent helped us brainstorm our design, and Bob Rodriguez kindly took time off from his summer vacation to help us understand the arcana of Ultrix. Chet Juszczak provided both encouragement and a reality check, and Dan Kolkowitz provided access to a Sun workstation. Andrew Black and the anonymous referees prompted many necessary changes, and Bill Hamburgen, Mary Jo Doherty, Joel McCormack, and Joel Bartlett helped to proofread the many drafts. Our mistakes are our own, of course.

References

[1] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh. A Remote-File Cache for RFS. In *Proc. Summer 1987 USENIX Conference*, pages 275-280. Phoenix, AZ, June, 1987.

[2] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* 6(1):51-81, February, 1988.

[3] Chet Juszczak. Improving the Performance and Correctness of an NFS Server. In *Proc. Winter 1989 USENIX Conference*, pages 53-63. San Diego, February, 1989.

[4] Christopher A. Kent. *Cache Coherence in Distributed Systems.* PhD thesis, Purdue University, 1986. Also available as Digital Equipment Corporation Western Research Laboratory Research Report 87/4.

[5] S. R. Kleiman. Vnodes: An Architecture for Multiple File Systems Types in Sun UNIX. In *Proc. Summer 1986 USENIX Conference*, pages 238-247. Atlanta, GA, June, 1986.

[6] 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 Communication* SAC-1(5):842-857, November, 1983.

[7] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems* 6(1):134-154, February, 1988.

[8] Michael J. K. Nielsen. *Titan System Manual.* Research Report 86/1, Digital Equipment Corporation Western Research Laboratory, September, 1986.

[9] John Ousterhout. Private communication. 1989.

[10] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10th Symposium on Operating Systems Principles*, pages 15-24. Orcas Island, WA, December, 1985.

[11] G. J. Popek and B. J. Walker, Eds. *The LOCUS Distributed System Architecture.* The MIT Press, Cambridge, MA, 1985.

[12] R. Rodriguez, M. Koehler, and R. Hyde. The Generic File System. In *Proc. Summer 1986 USENIX Conference*, pages 260-269. Atlanta, GA, June, 1986.

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network filesystem. In *Proc. Summer 1985 USENIX Conference*, pages 119-130. Portland, OR, June, 1985.

[14] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A Caching File System For A Programmer's Workstation. In *Proc. 10th Symposium on Operating Systems Principles*, pages 25-34. Orcas Island, WA, December, 1985.

[15] V. Srinivasan and Jeffrey C. Mogul. *Spritely NFS: Experiments with and Implementation of Cache-Consistency Protocols.* Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, March, 1989.

[16] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification.* RFC 1094, Network Information Center, SRI International, March, 1989.

[17] *Webster's New Collegiate Dictionary.* G. & C. Merriam Company, Springfield, MA, 1979.

[18] Brent B. Welch. *The Sprite Distributed File System.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California—Berkeley, 1989. In preparation.