Recovery in Spritely NFS

Jeffrey C. Mogul
Digital Equipment Corporation
Western Research Laboratory

ABSTRACT: NFS suffers from its lack of an explicit cache-consistency protocol. The Spritely NFS experiment, which grafted Sprite's cache-consistency protocol onto NFS, showed an improvement in NFS performance and consistency but failed to address the issue of server crash recovery. Several crash recovery mechanisms have been implemented for use with network file systems, but most are too complex to fit easily into the NFS design. Spritely NFS now uses a simple recovery protocol that requires almost no client-side support and guarantees consistent behavior even if the network is partitioned. This work demonstrates that one need not endure a stateless protocol for the sake of a simple implementation.

1. Introduction

NFS has been extremely successful, in large part because it is so simple and easily implemented. The NFS "stateless server" dogma makes implementation easy because the server need not maintain any (non-file) state between RPCs and so need not recover state after a crash.

Statelessness is not inherently good. Since many NFS operations are non-idempotent and might be retried due to a communication failure, to get reasonable performance and "better correctness" the server must cache the results of recent transactions [Juszczak 1989]. Such cache state is not normally recovered after a crash, although not recovering this state exposes the client to a possible idempotency failure.

A more serious problem with NFS statelessness is that it forces a tradeoff between inter-client cache consistency and client file-write performance. In order to avoid inconsistencies visible to client applications, NFS client implementations (by tradition, rather than specification) force any delayed writes to the server when a file is closed, thus ensuring that when clients use the sequence

```
Writer          Reader

open()
write()
close()
                open()
                read()
                close()
```

the reader will see the most recent data, if the writer and reader explicitly synchronize so that the reader's *open* takes place after the writer's *close*. (Actually, even this strategy doesn't quite work, as we will see shortly.)

Unfortunately, this means that the *close* operation is synchronous with the server's disk. Since most files are small [Baker et al. 1991; Ousterhout et al. 1985], this effectively makes most file writes synchronous with the server's

disk, and NFS clients spend much of their time waiting for disk writes to complete. Also, although many files have very short lifetimes, are never shared, and need never leave the client's cache, NFS still forces them to the server's disk and so wastes a lot of effort. Finally, NFS does not guarantee cache consistency for simultaneous write-sharing, because while a file is open, writes are not actually synchronous.

NFS implementations also trade cache consistency for client file-read performance. Because the client's cache would be useless if the client continually checked with the server to see if the underlying file has been modified, practical implementations only check every few seconds (this interval may vary based on how recently the file was last modified). Hence, if a client has cached an old version of a file, for several seconds after the writer's close the reader may see stale data. That is, NFS implementations do not even guarantee consistency in the sequential write-sharing case. The net result is that occasional consistency errors plague NFS users, yet NFS cannot aggressively use client caches to improve performance.

The Sprite file system [Nelson et al. 1988] solves this paradox by introducing an explicit cache-consistency protocol. The fundamental observation is that write-sharing is rare. It can be detected by the server if clients report file opens and closes (not done in NFS), so the write-through-on-close policy can be eliminated. Instead, when write-sharing does occur, Sprite turns off all client caching for the affected file, and thus provides true consistency between client hosts.

Spritely NFS [Srinivasan and Mogul 1989] was an experiment to show that a Sprite-like consistency protocol could be grafted onto NFS, and to show that the performance advantage of Sprite over NFS was in large part due to the consistency mechanism rather than other differences between Sprite and UNIX. Because the cache consistency protocol introduces server state that must be preserved across crashes, Spritely NFS (like Sprite) requires a crash-recovery protocol. This paper describes the design, implementation, and performance of a simple but robust recovery protocol. Even in the event of a network partition, no undetected consistency failures can occur.

The main ingredients of Spritely NFS and its recovery protocol are

- a superset of the NFS protocol, which allows the use of existing implementations and experience, and full interoperation with NFS clients and servers
- explicit cache consistency, providing guaranteed consistency and better performance
- server-centric recovery, which simplifies the client implementation and supports fast recovery after a server crash

- support for write-behind (asynchronous writes), which improves performance, including a technique that avoids undetected failures due to lack of server disk space
- detection and possible resolution of any failures caused by network partitions.

None of these individual concepts are entirely novel. The main contribution of the Spritely NFS project is the combination of these concepts to significantly improve upon NFS without adding excessive implementation complexity.

1.1. Status of the Project

The original Spritely NFS experimental work was done in 1988. At that time, we asserted (perhaps naively) that since the Sprite researchers had devised a recovery protocol for their system [Welch 1990], the problem of recovery was solved in principal for Spritely NFS as well. The Sprite recovery protocol, however, was not entirely satisfactory for Spritely NFS (see Section 5), and so during the next few years, several people[1] participated in a discussion of possible alternatives.

In early 1992, I finally worked out a relatively pleasing recovery design, based in large part on more recent work done on Sprite recovery [Mary G. Baker, personal communication, 1991]; this paper design was presented at a workshop [Mogul 1992]. That summer, an initial implementation of the recovery protocol was done by Bharat Shyam, an intern at my lab. Subsequently, I continued the implementation work to the point where the recovery protocol now works reliably and efficiently.

This paper, therefore, describes a system that is about three-quarters complete. Because it is embedded in an existing commercial implementation of NFS (ULTRIX, version 4.3), it may require some additional tuning to yield the best possible performance (some conservative assumptions made by the original NFS code are no longer necessary). Also, the cache-consistency protocol may enable several further improvements, such as directory caching and better file locking; these are described in Section 10. However, as it stands today, it works well enough to be used instead of NFS.

1. Mary G. Baker, Cary Gray, Rick Macklem, John Ousterhout, and Brent Welch.

2. *Goals and Design Philosophy*

The design of Spritely NFS, including its recovery protocol, is meant to meet a number of pragmatic goals:

- Simplicity: Spritely NFS was a successful experiment partly because it required minimal changes to an NFS implementation and almost no changes to any other code. Any improved version should avoid unnecessary complexity, especially on the client side; client hosts are often underpowered and administered by naive users.
- Consistency: Spritely NFS should provide guaranteed cache consistency at all times. A partial guarantee is no improvement on NFS, since an application cannot make use of a partially guaranteed property.
- Performance: Spritely NFS is not worth doing unless its performance, even with recovery, is better than that of NFS. Although Spritely NFS also promises better consistency, that in itself would not convince many users to switch.
- Reliability: Spritely NFS should be no less reliable than NFS or the local UNIX file system. (Note that I am satisfied with matching the lesser of these reliabilities in a given situation. NFS is sometimes, but not always, more reliable than a local UNIX file system, and Spritely NFS sometimes must give up these NFS properties.)
- No-brainer operation: System managers should not need to do anything special to manage a Spritely NFS system. In particular, they should not need to adjust parameter values. The timeouts in the system should reflect the delays inherent in network communication and should never have to be tuned to provide correct behavior in the face of slow hosts.
- Incremental adoption: Spritely NFS clients should interoperate with NFS servers, and vice versa. Otherwise, users will not have much of an incentive to adopt Spritely NFS, because they would have to replace large parts of their infrastructure all at once.

The system described in this paper meets these goals.

3. *Review of Spritely NFS*

This section summarizes the design of Spritely NFS, postponing the issue of recovery until Section 5. The Appendix gives a brief specification for the current version of the protocol, including recovery. Table 1 lists the RPCs added to the basic NFS suite and indicates where they are described in this paper.

Table 1. RPC Calls Added in Spritely NFS.

RPC name	Purpose	Described in
open	Inform server that client will use a file.	Section 3
close	Inform server that client reference to file is no longer needed.	Section 3
callback	Inform client that it must no longer cache a file.	Section 3
beginrecov	Inform client that server is recovering.	Section 6.3
endrecov	Inform client that server recovery is complete.	Section 6.3
reqreopen	Ask client to reopen some of its files.	Section 6.3
reopen	Inform server that client was using a file prior to the server's recovery.	Section 6.3
clctl	Inform server that client has rebooted or is trying to recover from a network partition.	Section 6.2, 7

The original Spritely NFS introduced two new client-to-server RPC calls, *open* and *close*. Both calls transmit the current number of read-only and read-write references the client has to the file; because only the client increments and decrements the reference counts, these RPCs are idempotent. (All Spritely NFS RPCs can be duplicated safely by the network, including those used in the recovery protocol. Those that are not idempotent employ a monotonically increasing sequence number, allowing the receiver to discard duplicates.)

The client data cache in Spritely NFS (as in Sprite and most NFS implementations) is organized as a set of fixed-size blocks, not as a whole-file cache. A Spritely NFS client maintains a flag per open file indicating whether blocks from that file may or may not be cached. If a file is not cachable, all reads and writes are done directly from or to the server. If a file is cachable, the client may use a cached copy of a data block, if present, to satisfy a read, and need not immediately write data through to the server.

The NFS server is augmented with a "state table," recording the consistency state of each currently-open file. In Spritely NFS, this state table is relevant only

to the *open* and *close* RPCs; all other client RPCs are handled exactly as in NFS. When a client issues an *open* RPC, the server makes an entry in its state table and then decides, based on other state table information, if the specified open-mode conflicts with uses by other clients. If the *open* is conflict free, the server (via the RPC return value) notifies the client that it can cache the file. Otherwise, the client is not allowed to cache the file.

In some cases, a conflict may arise after a client has opened a file and has been allowed to cache it. For example, the first client host might open a file for write, and be allowed to cache it, and then a second host might open the same file. At this point, in order to maintain consistency, the first client must stop caching the file.

For this reason, Spritely NFS adds a server-to-client *callback* RPC to the NFS protocol. When a server decides that a client must stop caching a file, it does a *callback* to inform the client. A client with cached dirty blocks may have to write these blocks back to the server before replying to the *callback* RPC.

Figure 1 shows what happens when the server detects write-sharing. (In the

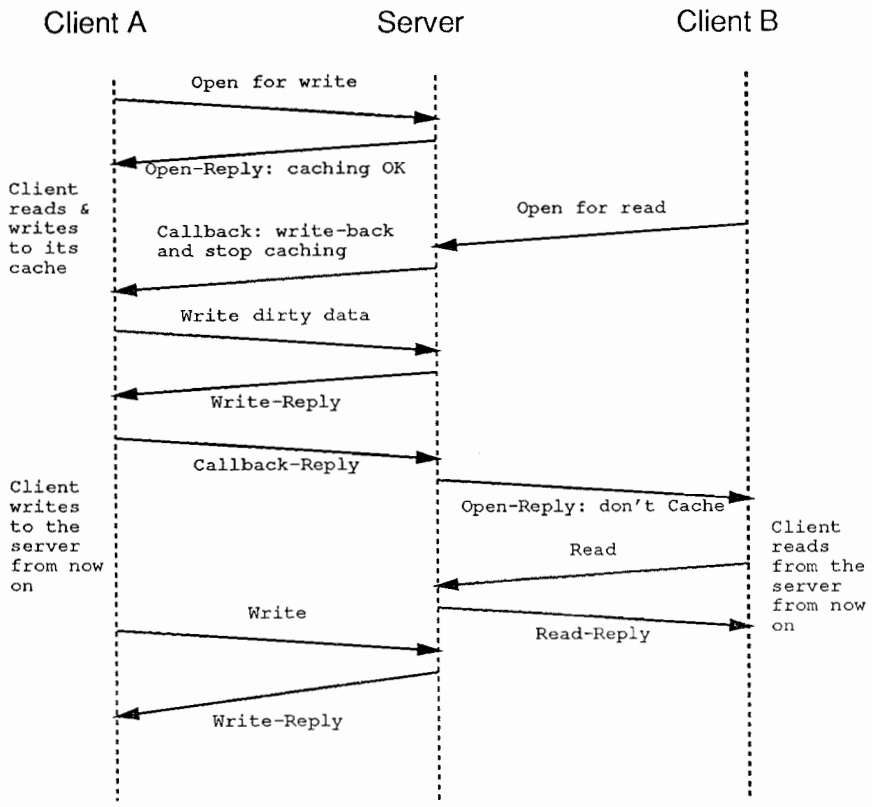

Figure 1. Time line for write-sharing situation.

figure, time flows down the page. Packets are shown as arrows; RPC requests are labeled above the arrow; and RPC replies are labeled below the arrow.) We see Client A open a file for writing, at which point the server allows caching (because no other clients are involved). Once Client B issues an *open*, however, the server does a *callback* RPC to Client A. In response, Client A writes back whatever dirty data it has, invalidates its cache, and disables further caching. It then replies to the server, at which point it is safe for the server to reply to Client B's *open* RPC. This reply informs Client B that it cannot cache the file; so from now on Client B will always read from the server (and Client A will always write through to the server).

Spritely NFS clients need not write-through dirty blocks when a file is closed. The server keeps track of closed-dirty files and can ask the client to write the blocks back if another client opens the file for reading, but otherwise the writer client can write the blocks back at its own leisure (see Figure 2; note that the use of two *close* RPC calls will be explained in Section 4). A client with closed-dirty blocks might even remove the file before the blocks are written back, thus

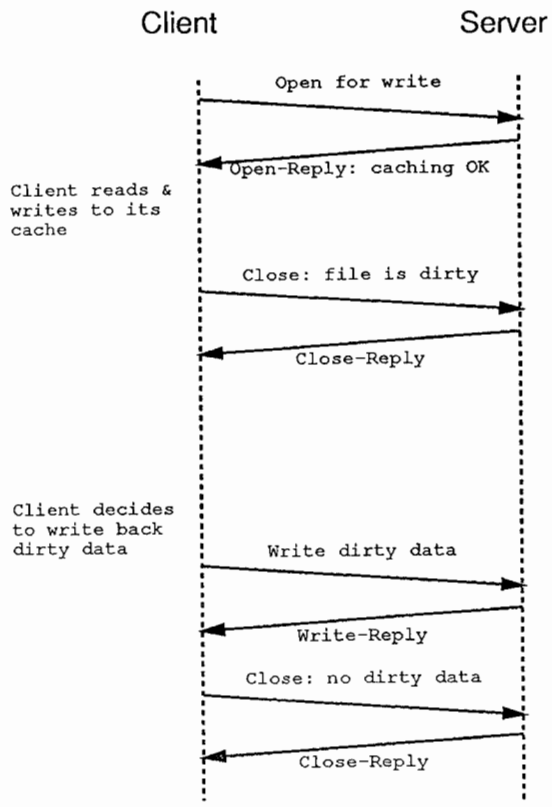

Figure 2. Time line for client using write-behind.

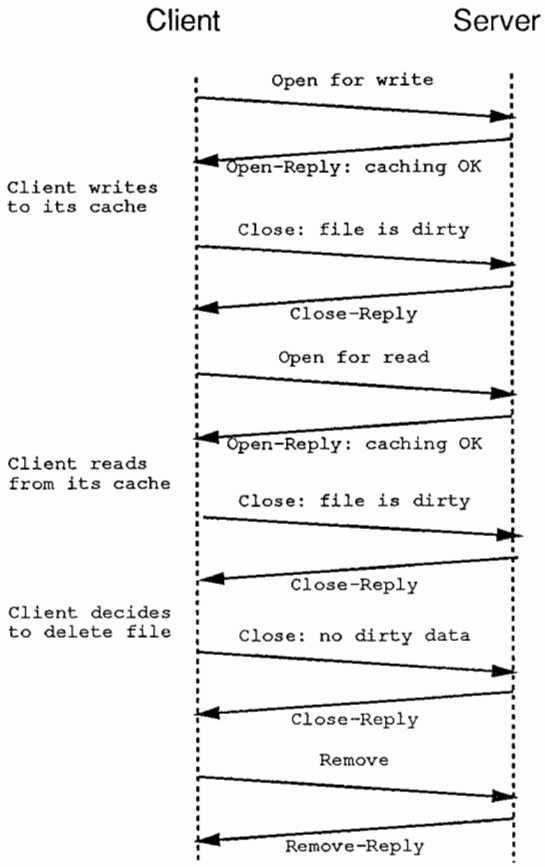

Figure 3. Time line for client removing file before write-back.

avoiding wasted effort (see Figure 3; note that no *write* RPCs are sent). We call this technique of writing dirty blocks after the file has been closed *write-behind.*

Write-behind does lead to one small semantic problem. The client does not know how many server disk blocks will be allocated to a file until all the data is written to the server, because the server's disk file system may require overhead blocks in addition to those used for data bytes. Commands that try to determine the storage allocation (such as *du* and *ls -s*) will underreport the value until the write-behind has completed. It might be possible to provide a close estimate; this is not done in the current implementation.

3.1. Performance Paradoxes of Write-Behind

Consistency in Spritely NFS (and Sprite) applies to entire files, not to regions within a file. Thus, if a client wants to read a file that is closed and dirty in

another client's cache, the server will not let the reader proceed until the writer has finished its write-back and has responded to the *callback* RPC. This may take a long time (on the order of minutes if the file is large, the writer has a large cache, and the server is slow).

This kind of delay does not occur with NFS (because the writer cannot keep many dirty blocks in its cache) or with a local file system (because there is no cache-consistency issue), and it may seem like a serious performance problem. It is, however, simply the exposure of a choice that is not necessary with a local file system and is not allowed by NFS.

Consider the example of a parallel *make* procedure, which first tells a number of client hosts to compile, in parallel, the individual modules making up the target program and then instructs one client to link the modules together. Relative to the object modules, the compiler hosts are writers and the linker host is the reader. Using NFS, the linker can start immediately after the last compilation is done, because at that point all the blocks of the object modules have been written to the server. If the clients use Spritely NFS and do write-behind, the linker will have to wait for all the write-backs to take place. In fact, it will probably open the object modules one at a time and so could be forced to pay the penalties serially.

Therefore, an application ought to have the ability to control when its writes are done. In the case of a parallel make, the compiler should specify that object-module writes take place asynchronously but as soon as possible, so that they will mostly overlap with other activity. (If the compiler includes an optimization pass that reads and rewrites the object file, only that pass should specify write-ASAP behavior, because the dirty data from the first pass need never go to the server.) During a single-host make, on the other hand, the compiler should specify write-behind because that will reduce the elapsed time until the compiled program is ready to use. This fact implies that the compiler must provide a way for the parallel-make program to instruct it to request write-ASAP behavior.

It might also be possible to devise heuristics that would allow the file system to decide whether write-behind or write-ASAP is appropriate. This decision could be difficult; one would not be able to rely on past accesses to the specific file, because (in the case of a parallel make) it has just come into existence; nor could one rely on the past behavior of a specific process because, in this example, a compiler process typically generates one object file and then exits. One possible approach would be to assume that files created in write-shared directories are likely to be shared themselves.

We could also limit the write-back delay by insisting that the writing client return the dirty blocks in sequence and by allowing the reading client to proceed with sequential reads as long as they did not get ahead of the writes. This method

would greatly complicate the protocol and its implementation and probably would not provide much advantage.

3.2. Cache Version Numbers

When a client reopens a closed file for which it has cached data blocks, how does it know if these cached blocks are still valid? We solve this problem by associating a version number with each file, which increases each time the file is opened for writing. The server returns the current and previous version numbers in reply to an *open* RPC; the client remembers the current version number for each file for which it might cache data blocks.

If the server's current version number matches the client's cached version number, the cached blocks are obviously current. If the server's previous version number matches the client's cached version number, and the client is performing an open-for-write, then the change in version number was caused by the current *open* operation, and the cached blocks are still current. Otherwise, some other client may have written the file since the cached blocks were read, and they must be invalidated.

Sprite maintains each file's version number on disk (i.e., in nonvolatile storage). For Spritely NFS, which makes use of existing disk file system designs, we decided that it would be infeasible to require the server to maintain a nonvolatile file version number. Spritely NFS still guarantees that a file's version number will increase on every open-for-write, as long as the server does not crash. During crash recovery, clients obtain new version numbers from the server for the files they have open (or for which they have cached dirty data), but not for files that are closed-clean. Hence, client caches of data for closed-clean files must be invalidated on server recovery. We believe that this will not seriously reduce the effectiveness of the client cache.

A Spritely NFS server stores a file's current version number in the corresponding state table entry. While the file is open, this entry is guaranteed to persist, but (because server memory is bounded) the state-table entry for a closed file may be kicked out to make room for a new active entry. If a client then reopens the file, the server no longer remembers the file's current version number. We preserve correct behavior in this case by assigning a new version number that is known to be higher than any previously issued, forcing the client to invalidate its old cached blocks even though they might be valid. This should not cause much performance loss unless the server's state table is nearly full.

The simplest way for the server to ensure that a file's version number increases, even if the file is not listed in any table, is to use a global counter and increment this counter whenever a file is opened for writing. If the server performs

an open-for-write once each millisecond (probably the rate would be much lower than this), a 32-bit counter would roll over after about seven weeks. At that time the server could simply invoke its crash-recovery mechanism to force the clients to obtain new version numbers.

The server could avoid incrementing the global counter on each open-for-write by incrementing instead only the values in active state-table entries. When a state-table entry is replaced (and hence its value is about to be lost), the global counter must then be updated to be the maximum of its current value and the value in the deleted entry. It could then never be less than the proper value for a "forgotten" entry. This would probably reduce the rate of counter wraparound, although the worst-case behavior would not change.

3.3. Choice of Update Policy

A system that delays writing data to the disk or file server should not delay these writes forever. A mechanism that writes out dirty blocks older than a certain age limits the potential damage caused by a crash and may also improve read latency by maintaining a pool of clean blocks.

The traditional UNIX update policy is to write out all dirty blocks every 30 seconds, along with all the modified metadata (that is, structural information about the file system, including superblocks and inodes) [McKusick et al. 1984]. This simple "periodic update" (PU) policy has been shown in a simulation study [Carson and Setia 1992] to increase read-latency (and especially the variance in read-latency), because at twice a minute the disk queues can become quite long. The same study suggests that the "individual periodic update" (IPU) policy, in which each block is written as its age reaches a threshold, should provide better performance (essentially by spreading the write load more smoothly).

In practice, it is simpler to approximate IPU by checking the age of dirty buffers every few seconds, rather than managing a precise timer for each dirty buffer. This means that, in the best case, one $N/30$ of the dirty blocks will be written every N seconds, which (for small N) keeps the variation in queue length much smaller. In the worst case, when the system manages to dirty the entire cache during a period of N seconds, IPU and PU perform identically.

Sprite originally used a modified version of the PU policy [Nelson et al. 1988]. It now uses a policy similar to IPU: Every five seconds, the system iterates over modified files and writes out all the cached blocks of any file whose oldest dirty block is at least 30 seconds old [John H. Hartman, personal communication, 1993].

The motivation behind this version of the IPU policy was not, apparently, to reduce read latency by reducing peak queue length but rather to maximize the

average write-back delay without increasing the worst-case exposure to crashes. Increasing write-back delay should improve performance if, as the Sprite designers believed [Ousterhout et al. 1985], many files have lifetimes so short that they need never be written out to the disk or server. (The original UNIX policy can write out blocks aged anywhere between 0 and 30 seconds and so will often write blocks from short-lived files.) Since the Sprite policy ignores the ages of the individual blocks, if a large file is written gradually into the cache, after about 30 seconds all of its blocks will suddenly be forced onto the output queue. Sprite's policy therefore approximates IPU only for small or rapidly written files.

The current implementation of Spritely NFS uses a simple mechanism that closely approximates IPU: Once a second, any dirty block older than 30 seconds is written out. As in the current Sprite method, a block's age is measured from the time it is first made dirty, rather than from the most recent modification.

Because NFS does not often delay writes, it is essentially insensitive to the update policy. Spritely NFS introduces delayed writes, and so it does benefit from an IPU policy, although the protocol does not specify any particular update policy. In Section 8.1, I will show that IPU provides a small but measurable performance advantage.

3.4. Avoiding Unbounded Timeouts during Callbacks

One aspect of the callback mechanism requires some ingenuity in order to avoid complex timeout issues. Because it may take a client an arbitrarily long time to respond to a *callback* (during this time, it may have to write a lot of data to an arbitrarily slow server), the server cannot infer from a timeout on a *callback* RPC that the client is down.

The solution to this predicament involves several tricks. First, we note that the RPC layer normally reissues a request several times before telling the caller that the request has timed out. Consider a client that, because it is busy doing the write-back, fails to reply in time to the first instance of a *callback* RPC. When the server retransmits the *callback*, the client notices that a *callback* is currently in progress for the specified file, and replies to the retransmitted RPC with a "try later" error code. The server, upon receiving this code, delays for a while and tries again. (The protocol might be more robust if it allowed the server to determine that the client is making "reasonable progress," to avoid deadlocks resulting from communications error.)

This mechanism could fail if the server's first few RPC requests really are dropped (by a lossy network or a busy client), and so the server uses an additional mechanism to avoid prematurely declaring a client to be dead. If the *callback* RPC times out, the server issues a *null* RPC, to which a live client should respond

immediately. If the client does respond, then the server retries the *callback* RPC (but with a longer timeout); otherwise, the server declares the client dead.

While the server is waiting for the *callback* to complete, the client that issued the *open* that caused the *callback* is also waiting for the server, and its RPC might also time out. We solve this by allowing the server to return the "try later" error code in response to a prematurely retransmitted *open*, which causes the client doing the *open* to delay for a while and then retry. Unless the reply to the original *open* is somehow dropped, the final retransmitted *open* is redundant, but because *open* is idempotent, no real harm is done.

Figure 4 shows an example in which both the server and the client doing the *open* must wait longer than their RPC timeouts while another client writes back its dirty data. This example represents more or less the worst case; the point to observe is that no matter how long the write-back takes, none of the participants will give up.

3.5. Automatic Recognition of Spritely NFS Hosts

I argued that without a path for incremental adoption, users will have little incentive to install Spritely NFS, because all-at-once changeovers cause major disruption. A sudden change to a new, untried system makes system managers nervous.

Spritely NFS uses the same RPC program number and version as NFS and can easily coexist with pure NFS hosts. The two problems to solve are automatic configuration (so that network managers need not worry about who is running what) and maintenance of consistency guarantees (so that NFS clients get at least the level of consistency that they would if all clients were using NFS).

When a Spritely NFS client mounts a file system from a remote server, the first RPC that it issues (after having gone through the mount protocol) is a *clctl* (client-control) RPC. (The details of this RPC are related to crash recovery and are discussed in Section 6.2.) If the server speaks only NFS, it will respond to this with a PROC_UNAVAIL error code. The client records this fact (in a per-filesystem data structure) and treats the file system as a purely NFS service. If the server responds to the *clctl* RPC, then it must be a Spritely NFS server, and the client then follows the Spritely NFS protocol.

A Spritely NFS server recognizes Spritely NFS clients because they issue *clctl* RPCs before anything else. Spritely NFS servers keep track of the addresses of their Spritely NFS clients, to be used for crash recovery (see Section 6.3) and space reservation, but otherwise act identically to NFS servers.

In principle, we can establish rules that allow Spritely NFS hosts to detect when their peers change flavor (i.e., upgrade from NFS to Spritely NFS or change

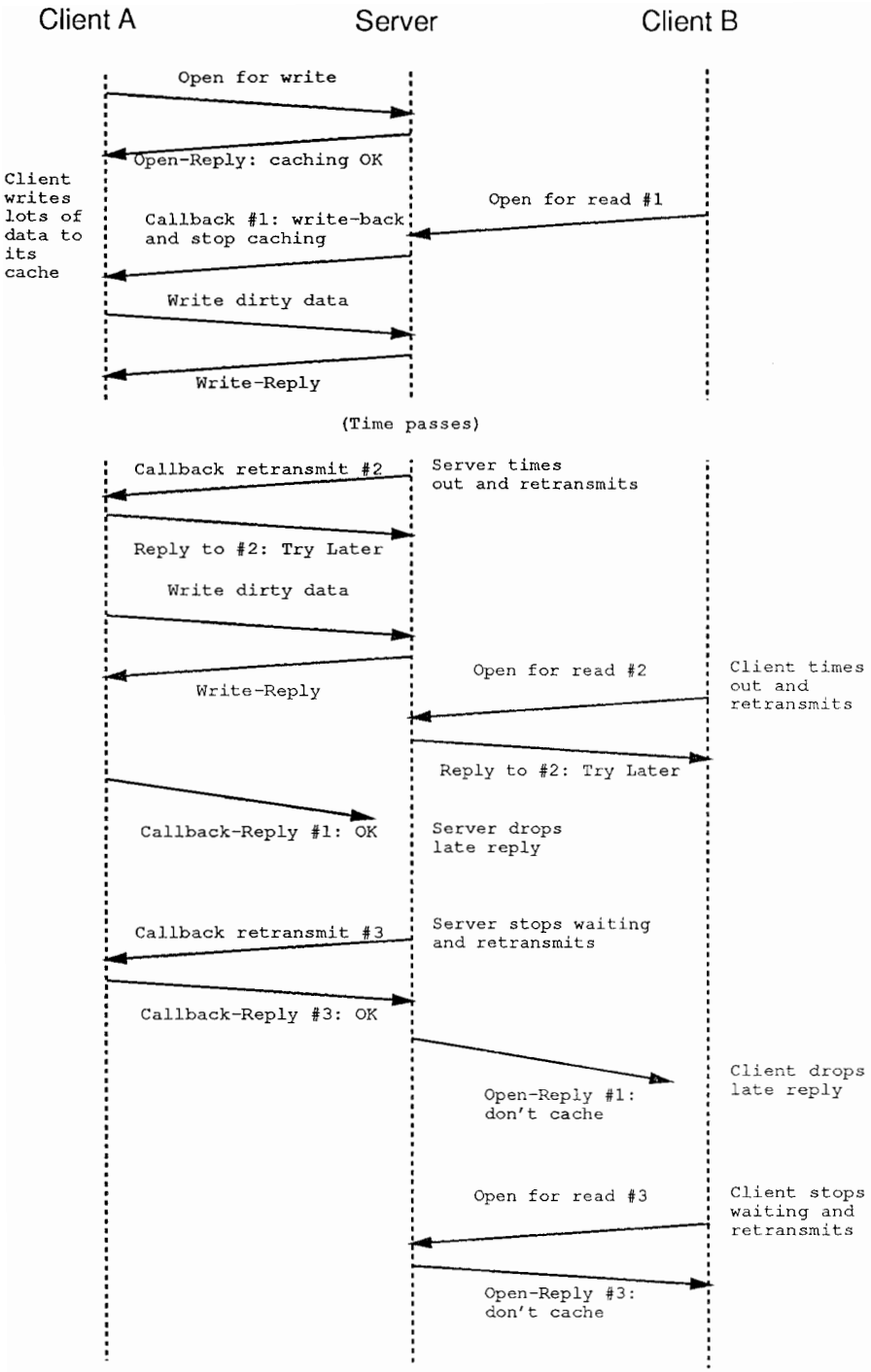

Figure 4. Time line for timeouts during *callback*.

from Spritely NFS back to NFS). Some speculations on how this might be done were presented in an earlier paper [Mogul 1992].

3.6. Consistency between Spritely NFS and NFS Clients

A Spritely NFS server should provide consistency between NFS and Spritely NFS clients write-sharing a file (as much as possible). The server does this by treating each NFS RPC referencing a Spritely NFS file as if it were bracketed by an implicit pair of *open* and *close* operations. (The actual implementation should be more clever, bypassing the Spritely NFS state transition machinery when possible, to avoid adding excess delay to NFS operations. Also, it should not attempt to send a *callback* to an NFS client that has a file "open.")

This gives Spritely NFS clients nearly perfect consistency (since NFS clients use write-through, more or less). The NFS clients get no-worse-than-NFS consistency, since any reads they do to the server return the latest copy of the data. (NFS clients cannot get perfect consistency, for they sometimes read from their caches when their caches are stale.)

An NFS client attempting to access a file for which a Spritely NFS client is currently caching a lot of dirty data might run into the kind of delay described in Section 3.1. That is, the Spritely NFS client might take a long time to write back its dirty data; so the server might not immediately respond to the NFS *read* or *write* RPC.

In Section 3.1, this was described as a performance issue, because Spritely NFS clients are willing to wait indefinitely for an *open* to complete. With an NFS client, however, we cannot use the mechanism described in Section 3.4, returning a special error code saying "try again later," because NFS clients will not recognize this code. The upshot is that a NFS client using a "soft mount" (which turns RPC timeouts into failures) could report a transient error when reading or writing a file being written by a Spritely NFS client. This can happen only if the file is large and the Spritely NFS client has written it quickly, causing a sizable backlog of dirty blocks to accumulate in its cache.

3.7. Evolution of the Spritely NFS Protocol

The Spritely NFS protocol has evolved somewhat since our original publication [Srinivasan and Mogul 1989]. In addition to the recovery mechanisms, we found that we had to make several changes to the cache-consistency protocol itself.

The original specification for the *open* RPC, instead of passing read and write reference counts, passed a boolean flag indicating whether this new reference

would be used for writing. The server was responsible for increasing the reference counts as each *open* was received. This design was flawed in that it was not idempotent; a duplicated RPC would cause the server's counts to be wrong. A race between a duplicated *open* RPC and a *close* RPC could cause serious problems. In the new design, the client maintains the reference counts and simply reports them to the server. The old design meant that we could not use a single method for both *open* and *reopen*, because *reopen* inherently needs to pass reference counts.

Similarly, the *close* RPC originally passed a boolean flag indicating if the reference being closed had been open for writing. We replaced this with a pair of reference counts. We also added a count of the number of cached dirty bytes to support the space-reservation mechanism.

We also added the possibility for the *open* and *callback* RPCs to return a new status code, meaning "try again later." This strategy avoids the need to bound the amount of time needed for a client to write back its dirty blocks, as a result of a *callback* (which in turn is always the result of an *open*). Instead, all the parties involved now know to keep retrying an operation until it succeeds, fails, or times out. A timeout now indicates a host or network failure and cannot be the result of having too much work to do in a fixed amount of time.

This technique would not have been practical without the change to an idempotent *open* RPC. Using the old protocol, it would have been tricky to keep track of the half-completed *open* operations to avoid incorrectly increasing the reference counts. Also, the new protocol avoids the need to lock a state table entry while a *callback* is in progress; this eliminates a possible deadlock if the called-back client decides to close the file as a result of the *callback*.

4. Dealing with ENOSPC

One problem with the write-behind policy is that one or more of these writes might fail. In NFS, because the client implementation forces all writes to the server before responding to the *close* system call, an application which checks the return value from both *write* and *close* calls will always know of any write failures. Not so in Spritely NFS, because the failure might occur long after the application has released its descriptor for the file (or even after the application has exited). This could cause trouble for applications that do not explicitly flush their data to disk.

Three categories of error can occur on a client-to-server *write* operation:

1. Communications failure: The network is partitioned or the server crashes, and the RPC times out before the failure is repaired.
2. Server disk hardware error: The disk write operation fails, or the disk fails after the write completes.
3. Server out of disk space: No space is available on the server disk.

The first error can be turned into a delay by simply retrying the RPC until the server responds.[2] If the client crashes in the interim, then the dirty block is lost, but this situation is no different from a normal local-filesystem delayed write in UNIX.

The second error is not generally solvable, even by a strict write-through policy. It is true that the NFS approach will report detectable write failures, but these are increasingly rare (because techniques such as bad-block replacement can mask them). Again, normal UNIX local-filesystem semantics does not prevent this kind of error from occurring long after a file has been closed.

The third error (ENOSPC, in UNIX terms) is the troublesome one. We want to report this type to the application, because it might want to recover from the condition and because there is no obvious way for the underlying file system mechanism to recover from ENOSPC. (Also, unlike the other two kinds of errors, one cannot avoid ENOSPC errors through fault-tolerance techniques.)

Sprite does not completely solve this problem; that is, Sprite applications can believe their writes are safe but the delayed writes pile up in a volatile cache because the server is out of space [Mary G. Baker, personal communication, 1992]. AFS apparently follows the same approach as NFS, forcing modified data back to the server when the file is closed [Kazar 1988].

Spritely NFS solves the ENOSPC problem by reserving disk space for the remaining dirty data when the file is closed. That is, when a dirty file is closed, the client counts up the number of dirty bytes and requests that the server reserve that much disk space for the file. The server may respond with an ENOSPC error at this point, in which case the client can revert to a write-through on close policy. Note that the server may respond to *close* with ENOSPC even when enough space does exist; so the client must attempt the writes and report an error to the application only if a *write* actually fails.

A client cannot exactly determine the amount of space required to write a set of dirty buffers. Server file system space is typically allocated in units of blocks,

2. This point is not true if the client uses a "soft mount," which turns RPC timeouts into errors rather than retries. Soft mounts are generally thought of as living dangerously, although delaying writes after a *close* does make them even more dangerous. Perhaps soft-writes-after-close should be made "harder" as long as the client has enough buffer cache to avoid interference with other operations.

not bytes. NFS does provide the block size to the client; so the client's reservation request conservatively rounds up the size of every dirty buffer to a multiple of the block size. Also, because the underlying disk file system may require overhead space (such as "indirect blocks"), the server must increase the reservation by its own conservative estimate of the number of overhead bytes.

Because the space reservation is a conservative overestimate, when the client is done writing back the dirty data, some residual space will probably still be reserved at the server. Also, a client might truncate or remove a file before writing back the dirty buffers. At some point the client must release the excess reservation, or else the server's disk would gradually become committed to phantom data. The client does so by sending a final *close* RPC, which "reserves" zero bytes for the file (see Figures 2 and 3). This also causes the server to change its state-table entry from closed-dirty to closed, which allows the table storage to be reclaimed.

The bookkeeping for the space reservation made by a *close* RPC involves maintaining two separate counts. The Spritely NFS server layer keeps track of the remaining reservation for the given file in the corresponding state-table entry. It must also arrange with the underlying disk file system to reserve some of the remaining free space for the file in question. The file system need not actually allocate space on disk for the reservation; rather, it only needs to keep a count of the number of free bytes and the number of reserved bytes, and ensure that the difference never becomes negative. The reservation counts need not be kept on stable storage, since they will be recovered during the server crash-recovery phase (see Section 6.6). Thus, the overhead of maintaining these counts is quite low. Note that the file system must enforce the reservation against local applications as well as against remote clients.

When a server handles a *write* RPC for a closed file, it decreases the reservation in the state-table entry and must also tell the underlying file system to decreases its count of reserved bytes. (If a client in the closed-dirty state tries to use more space than its reservation allows, the *write* will fail.) Both counts are decreased by the amount of new space actually allocated as a result of the *write*, not the transfer count of the *write* RPC (which may be either larger or smaller than the actual new space required).

This procedure is difficult to implement correctly in the highly layered ULTRIX server code (typical, probably, of other NFS implementations) because there is no straightforward way to determine how much space is required for a file-system write without actually performing the write. The current implementation makes this determination by assuming the worst possible space allocation, making this space available for writing by deducting it from the reservation, performing the write, and then seeing how much space was actually allocated. The difference between the worst-case value and the actual value (i.e., the amount of

reservation that was deducted before the write but not used) is then added back to the remaining reservation. This results in correct bookkeeping and does not require any restructuring of the traditional disk file system code, but it does leave a brief window where another writer could "steal" the last remaining blocks of free space on a disk. A race-free solution to the problem seems feasible but will require a much deeper understanding of the disk file system implementation.

One subtle problem can occur with this scheme if two processes on one client are writing the same file. After one successfully closes the file (i.e., the server grants a reservation), if the other process extends the file so much that the server runs out of disk space, some part of the file might not be written to the server. This example is not entirely contrived; the file might be a "log," appended to by multiple processes. The client implementation could preserve correct semantics in such a case by ordering the disk writes so that none of the blocks dirtied after the *close* are written to the server before the other dirty blocks of that file. The current client implementation simply forces all dirty blocks to the server before reopening a closed-dirty file for write; this is correct but wastes the benefit of write-behind for frequently opened files.

If a client crashes while holding a reservation, or simply never makes use of it, the space could be tied up indefinitely. Thus, the server should set a time limit on any reservation grant (perhaps in proportion to the number of blocks reserved; if a client reserves space for a billion bytes, it is unlikely that they could all be written back within a short interval. A server might also refuse to honor a reservation for more than a few seconds' worth of disk writes). When the time limit expires and if space is low, the server can reclaim the reservation by doing a callback (to force the client to write back the dirty blocks).

A client that fails to respond to the callback, perhaps because of a network partition, might end up being unable to write dirty blocks if the server reclaims its reservation. Because a partition might last arbitrarily long, there is not much that can be done about this: conceptually, this is the same as partition during a consistency-callback; in either case the write-caching client is unable to write its modifications back to the server. Section 6.1 will discuss how the protocol deals with such contingencies. To avoid unnecessarily provoking this problem, a server should refrain from reclaiming timed-out reservations as long as sufficient free space remains.

If, after the partition heals, the server has sufficient disk space and has not allowed a conflicting open, the client could transparently recover from the partition (see Section 7). On the other hand, if recovery is impossible, because no disk space is left or because conflicting access has been allowed, then the client host may have no way to notify the application that wrote the file. The application has already closed the file and may even have exited. In this case, the modified data

would be lost, just as if a disk sector had been corrupted; applications that cannot afford data loss should be taking measures to defend against it, whether caused by local-disk or network failure.

The space-reservation mechanism, in summary, appears to provide the correct failure semantics without seriously compromising performance. The change to the protocol is quite simple; however, this feature does complicate both the client and server implementations in a number of ways and for that reason is not entirely satisfying.

5. *Overview of the Recovery Protocol*

Several different recovery mechanisms might have been used for Spritely NFS. The original recovery mechanism used in Sprite [Welch 1990] depends on a facility implemented in the RPC layer that allows the clients and servers to keep track of the up/down state of their peers. When a client sees a server come up, the Sprite file system layer then reopens all of its files.

This approach provides more general recovery support than is needed for Spritely NFS, and it has several drawbacks. First, it would require changes to the RPC protocol now used with NFS, some additional overhead on each RPC call, and some additional timer manipulation on the client. In other words, it complicates the client implementation, which is something we wish to avoid. Second, recent experience at Berkeley [Baker and Ousterhout 1991] has shown that such a "client-centric" approach can cause massive congestion of a recovering server. Sun RPC has no way to flow-control the actions of lots of independent clients (a negative-acknowledgement mechanism was added to Sprite's RPC protocol to avoid server congestion [Baker and Ousterhout 1991]). Third, the server has no way of knowing for sure when all the clients have contacted it; even if all the clients actually respond quickly, the server still must wait for the longest reasonable client timeout interval in case some client has not yet tried to recover. This requirement can make fast recovery impossible. Fourth, if a partition occurs during the recovery phase, partitioned clients may never discover that they have inconsistent consistency state.

Another possible approach is the "leases" mechanism [Gray and Cheriton 1989]. A lease is a promise from the server to the client that, for a specified period of time, the client has the right to cache a file. The client must either renew the lease or stop caching the file before the lease expires. Since the server controls the maximum duration of a lease, recovery is trivial: once rebooted, the server simply refuses to issue any new leases for a period equal to the maximum lease duration. A server will renew existing leases during this period; the clients

will continually retry lease renewals at the appropriate interval. When the recovery period has expired, no old lease can conflict with any new lease, and so no server state need be rebuilt.

The problem with leases is that they do not easily support write-behind. Consider what can happen if a client holding dirty data is partitioned from the server during the recovery phase (not an unlikely event, since a network router or bridge might be knocked out by the same problem that causes a server crash), or if the server is simply too overloaded to renew all the leases before they expire. In either case, the client is left holding the bag: the server will have honored its promise not to issue a conflicting lease but will not have given the client a useful chance to write back its dirty data before a conflict might result.

Another potential problem with leases is that the duration of a lease is a parameter that must be chosen by the server. The correct choice of this parameter is a compromise between the amount of lease-renewal traffic and the period during which a recovering server cannot issue new leases, and it is unlikely that the average system manager will be able to make the right choice. The original Sprite protocol has a similar parameter, the interval between "are you alive?" null RPCs, which again trades off extra traffic against the duration of the recovery phase. We would like to avoid all unnecessary parameters in the protocol; these force people to make choices that might well be wrong. (Also, timer-based mechanisms force increased timer complexity and overhead on the client.)

Spritely NFS uses a "server-centric" mechanism, similar to one implemented for Sprite, that relies on a small amount of nonvolatile state maintained by the server [Mary G. Baker, personal communication, 1991]. The idea is that in normal operation, the server keeps track of which clients are using Spritely NFS; during recovery, the server then contacts these clients and tells them what to do. Since the recovery phase is entirely controlled by the server, there is less chance for congestion (the server controls the rate at which its resources are used). More important, the client complexity is minimal: rather than managing timers and making decisions, all client behavior during recovery is in response to server instructions. That is, the clients require no autonomous "intelligence" to participate in the recovery protocol.

For this proposal to work, the use of stable storage for server state must be quite limited, both in space and in update rate. The rate of reads need not be limited because a volatile cache can satisfy those with low overhead. Stable storage might be kept in a nonvolatile RAM (NVRAM), but if the update rate is low enough it is just as easy to keep it in a small disk file, managed by a daemon process. Updates to this disk file might delay certain RPC responses by a few tens of milliseconds, but (as you will see) such updates are rare.

6. Details of the Recovery Protocol

The stable storage used in this protocol is simply a list of client hosts, with a few extra bits of information associated with each client. One is a code saying whether a particular client is an NFS client or a Spritely NFS client. Only Spritely NFS clients participate in the recovery protocol, but we keep a list of NFS clients because it could be useful to a system manager. Another flag records whether the client was unresponsive during a recovery phase or callback RPC, allowing us to report to the client all network partitions, once they are healed.

6.1. Normal Operation

During normal operation the server maintains the client list by monitoring all RPC operations. If a previously unknown client makes a *clctl* RPC, then it is obviously a Spritely NFS system. If a previously unknown client makes any file-manipulating RPC, then it is an NFS client. If a host thought to be an NFS client does a *clctl*, then it presumably has become a Spritely NFS client and will participate in the Spritely NFS protocol from now on (see Section 3.5).

The client list changes only when a new client arrives (or changes between NFS and Spritely NFS). Such a change is an extremely rare event (most servers are never exposed to more than a few hundred clients), and so it does not matter how expensive it is. In the current implementation, the client list is kept in a disk file and the update cost is a few disk accesses, that is, comparable to the basic cost of a file access.

On the other hand, the server must check the cached copy of the client list on almost every RPC. This check is done quite cheaply by keeping the client list as a broad hash table and by keeping a one-entry look-aside buffer (because in many cases, the server will receive several RPCs in a row from the same client). The overhead should be less than is required to maintain the usual NFS transaction cache.

Note that the server's volatile copy of the client list need not contain the entire list of clients but could be managed as an LRU cache, as long as it is big enough to contain the working set of active clients. This approach might conserve memory if there are a lot of inactive clients on the list.

If a client fails to respond to a callback (or during the server recovery phase, described in Section 6.3) then the server marks it as "embargoed." This failure could occur because the client has crashed, but it might be because the client has been partitioned from the server. As described in Section 3, the server goes to

some lengths to ensure that a callback does not time out if the client is actually alive and reachable.

When an embargoed client tries to contact the server, the server responds to the RPC with an error code saying "you are embargoed." The client thus knows that it was partitioned during an operation that might have left its state inconsistent and can take action to repair things (or at least report the problem to the user). See section 7 for more details.

6.2. Client Crash Recovery

When a client crashes and reboots, we do not want to leave the server believing that the client has files open, which could lead to false conflicts and thus reduced caching. The server could discover some false conflicts if, when it does a callback, the client replies "but I don't have that file open." In other cases the false conflict would not cause a callback (i.e., if caching is already disabled) and would not be discovered. Also, these "false opens" waste resources, namely, entries in the server's state table and reserved space on the local file systems.

Spritely NFS solves this problem by using the *clctl* RPC, issued by a Spritely NFS client when it mounts a file system, to detect client reboots. The *clctl* RPC arguments include an op-code that in this case indicates that the client might have rebooted and an epoch number that allows the server to determine if a reboot has actually occurred. The client must generate an epoch number that increases monotonically on each reboot and otherwise does not increase at all. The server records the client's epoch number in the corresponding client list entry. If the epoch received in a *clctl* RPC is greater than the previously recorded epoch, the server closes all Spritely NFS files opened by that client and releases all associated space reservations.

The simplest way for a client to generate a monotonically increasing epoch is to use the time at which it booted. Most computer hardware now includes a battery operated clock, but the client could also use any of several simple time-server protocols to obtain the current time upon booting. If the client uses a clock with a resolution of one second, this means that it cannot reboot more than once per second, which is not likely to be an onerous restriction.

If the client uses NVRAM to keep dirty blocks across a crash, it should also use it to preserve a list of all writable file handles. When rebooting, it must first write back the dirty blocks, and then it can send the *clctl* RPC. Alternatively, the protocol might be modified to add a new *clctl* op-code that says "close all the files for which I have read-only references," allowing the client to write-back and close the writable files in the normal way.

If a client reboots while a server is down (or unreachable because of a network partition), the client simply keeps retrying its mount operation until the server recovers (or becomes reachable), just as is done in NFS.

When a client reboots and sends a *clctl* specifying "close all my open files," the server should also clear an embargo against the client (if one has been set). This is because a client starting with a tabula rasa does not care about the consistency state of files it previously had open.

6.3. Server Crash Recovery

When a server crashes and reboots, it first obtains the client list from stable storage and reloads it into volatile memory. The server then enters a recovery phase consisting of several steps, herding the clients through these steps by issuing a series of recovery-specific *callback* RPCs. The steps, diagramed in Figure 5, are

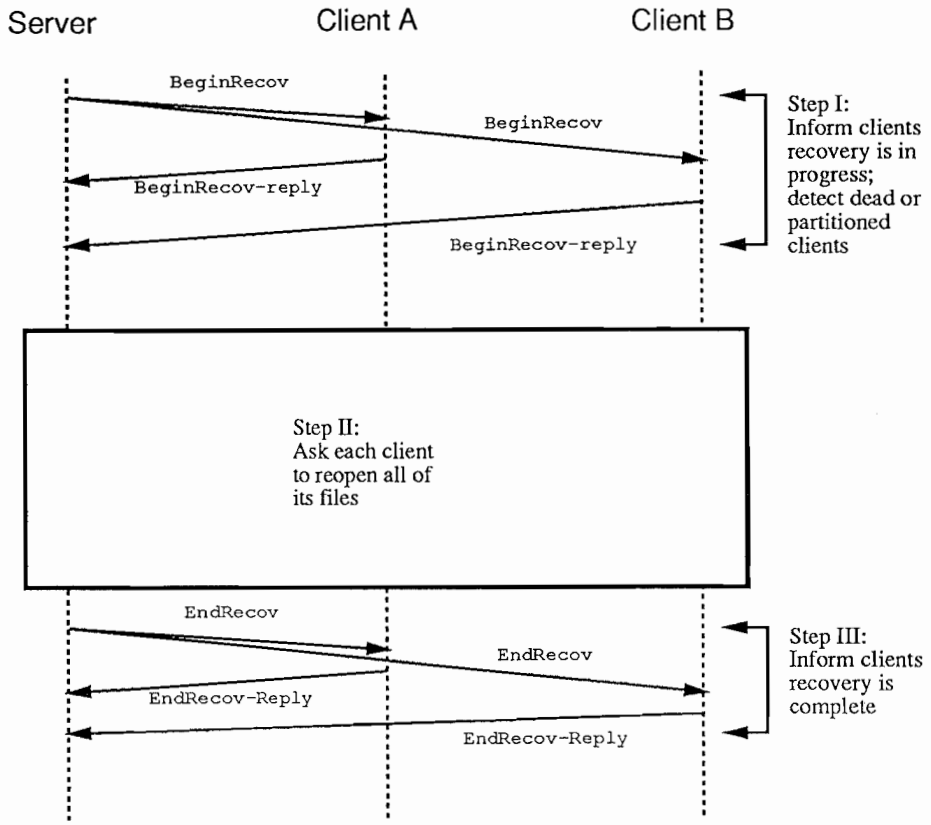

Figure 5. Time line for server recovery.

1. Initiate recovery: The server contacts each nonembargoed Spritely NFS client on the client list. The *beginrecov* callback RPC tells the client that the recovery phase is starting; until the recovery termination step is complete, clients are not allowed to do new *open* or *close* RPCs and cannot perform any data operations on existing files.

   When a client responds to this RPC, the server knows that the client is participating in the recovery protocol; clients that do not respond are marked as embargoed. During the rest of recovery, embargoed clients are ignored, and we can assume that the other clients will respond promptly. However, during this step long timeouts may be needed. On the other hand, *beginrecov* can be sent to all clients in parallel, so the worst-case length of this step is only slightly longer than the maximum timeout period for a single client.

   At the end of this step, we can update the stable-storage client list to reflect our current notion of each client's status.

2. Rebuild consistency state: The server contacts each nonembargoed client and instructs it to reopen all of the files that it currently has open, using the *reqreopen* (Request Re-open) RPC. If the clients do not cheat, the resulting opens will have no conflicts, because before the server crashed there were no conflicts, and no new opens could have taken place since the crash. See Figure 6 for an example of this step.

   Since each server may have to open multiple files, and file-open operations are moderately expensive (requiring manipulation of the state table), the server may want to do these callbacks serially rather than in parallel (or semiparallel, to limit the load to a reasonable value). This should not result in too much delay, for we are reasonably sure that the clients involved will respond.

   Files are reopened with the *reopen* RPC, which is similar in form to the *open* RPC except that it allows several files to be opened in one call. Also, in addition to conveying read-only and read-write reference counts for each file, it also conveys the number of dirty bytes for closed-dirty files, allowing the server to recompute space reservations. (See Section 6.6 for more details.)

   During this step, a client may have to reopen an arbitrary number of files, and so the server cannot bound the amount of time required to finish the process. Therefore, the *reqreopen* RPC authorizes the client to issue only a small number of *reopen* RPCs. The client's response to *reqreopen* includes a flag saying "I'm done" or "I have more files to reopen"; the server loops issuing the *reqreopen* RPC until the client responds "I'm done." In

Figure 6, Client A has 11 files to reopen and must reopen them in two separate batches; Client B has no files to reopen and immediately replies "I'm done."

Thus, the server can use the normal RPC timeout mechanism to decide if a client has died or become unreachable. If a client fails to respond to *reqreopen*, then the server marks it as embargoed and updates the stable-storage list. Otherwise, once the client responds "I'm done," the server is sure that all the client's files have been reopened.

At the end of this step, the server has a complete and consistent state table, listing all of the open and closed-dirty files. In the current version of the protocol, closed-clean files are not reopened, because that is not necessary for correctness.

3. Terminate recovery: The final step is to inform each client that recovery is over. Once a client receives the *endrecov* RPC, it can do any operation it wants. As in the recovery initiation step, the server can do these callbacks in parallel, but in any case because the clients are unlikely to time out, the duration of this step should be brief.

Note that NFS server hosts typically export a number of file systems, and a client may mount several file systems from a given host. All Spritely NFS recovery operations refer to hosts and files, not specific file systems. This approach works fine, and is simpler than the alternative, which would be to do recovery on a file system-by-file system basis.

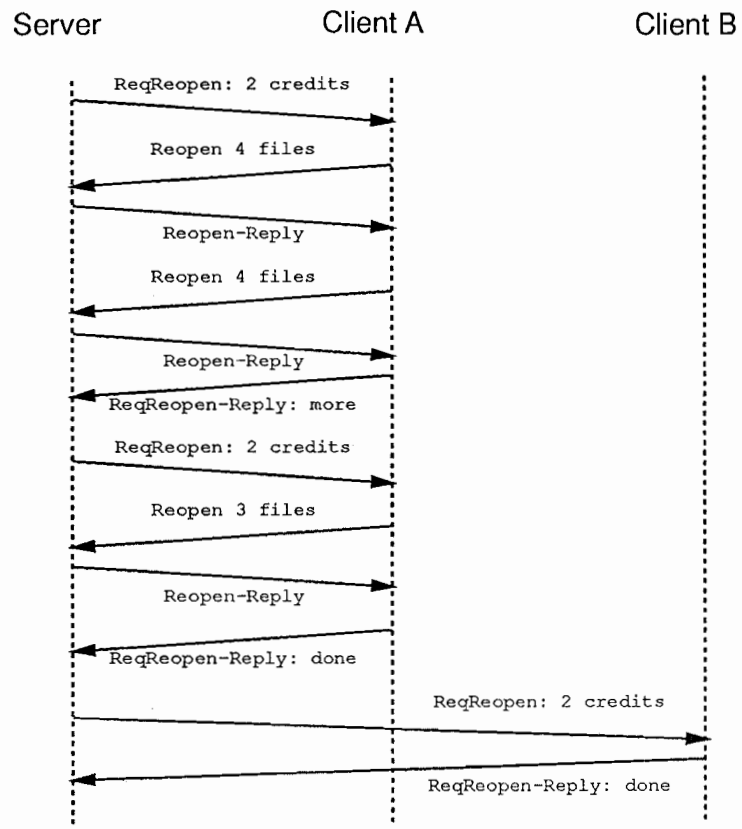

Figure 6. Time line for Step II of server recovery.

6.4. Crashes during Server Recovery

It is possible that the server may crash and reboot during the recovery phase. For this reason, we need a mechanism to ensure that the client reopens all of its files following the most recent reboot. To support this, the *beginrecov* and *endrecov* RPCs carry a monotonically increasing sequence number. The client simply ignores all stale and duplicated *beginrecov* and *endrecov* RPCs.

The condition that these sequence numbers are monotonically increasing must hold true across server crashes. One way for the server to generate the sequence number safely is to start with the time-of-boot, and increment the counter as subsequent *beginrecov* and *endrecov* RPCs are generated. (A system manager may want to trigger the recovery process even if the server has not crashed, in cases where the server's state may have become damaged.) As with the client epoch value, if the server uses a clock with a one-second resolution, the server cannot successfully reboot more than once every two seconds.

When the client receives a *beginrecov* with a new sequence number, it marks all of its open (or closed-dirty) files for that server as in need of reopening. When a *reqreopen* RPC arrives, the client searches its data structures for files still marked as in need of reopening. Thus, if the server crashes and reboots at any stage in this process, the client is assured of reopening all of its files before recovery ultimately terminates.

If a client crashes and reboots during server recovery, then of course it need not reopen any files. The server detects that the client has revived when the client sends its *clctl* RPC.

6.5. Log-Based Recovery

Because the recovery protocol is server-centric, it leaves the implementor of the server considerable freedom to choose different strategies. V. Srinivasan [Srinivasan, personal communication, 1992] has pointed out that nothing in the protocol prevents the server from using additional stable storage to obviate part or all of the recovery protocol. The server could, for example, log all opens and closes to stable storage. Since the "open lifetime" of files is fairly short (often less than 100 milliseconds [Baker et al. 1991]), it would not make sense to log every such event to disk. Instead, the server could keep the head of the log in NVRAM, which would allow it to elide the short-lived opens before writing the log to the disk.[3] Some sort of log-cleaning algorithm, analogous to that required by a log-structured

3. Or it could keep the entire state table in a huge NVRAM [Baker and Sullivan 1992]; see Section 11.

file system [Rosenblum and Ousterhout 1991], would be necessary. Alternatively, the on-disk information could be structured as a database, which would take more work to update but would not need cleaning. Using the log or database, the server could recover its consistency state without any help from the clients.

A much simpler approach would be to keep track, in the client list, of those clients that have any files open at all. During crash recovery, the server could ignore any client known to have no open files, thus speeding recovery and perhaps avoiding timeouts for clients that have been removed from service. This modification would increase the update rate for the stable-storage copy of client list. However, the server could delay the update on a client's last close, anticipating a subsequent open in the near future, because this delay would not affect the correct behavior of the recovery protocol. A delay interval of, say, one minute would probably avoid almost all extra updates without significantly increasing the cost of recovery.

6.6. Recovery of Space Reservations

Because the server host's count of reserved disk blocks may be updated quite often, it does not make sense to keep it in stable storage. (Maintaining a stable accurate value could approximately double the latency of disk writes for closed-dirty files.) Instead, we recompute this value during the recovery phase. When recovery starts, the server sets the value at zero. The clients then tell the server what their remaining reservations are, using the *reopen* RPC. Once recovery is done, the server has a consistent count of the total reservation requirements.

During recovery, a client cannot simply request a reservation for the number of dirty blocks it currently holds, because the server might have denied the reservation when it was initially requested. The client must remember when a *close* failed to obtain a reservation for a file, and when reopening that file must not request a reservation for it.

Note that it is not necessary for a client to compute how much of the requested reservation actually remains to be used, because no new application-level writes can be made to the file (even by the same client) while a client holds a reservation. In order to make new writes, the client must first reopen the file, which voids any pending reservation.

If the network is partitioned during recovery, we might end up in a state in which the server does not know of a client's reservation requirements and so gives the space away once recovery is over. If the partition heals, we may discover that no conflicting open prevents the embargoed client from writing its dirty blocks, but there is no longer any space to hold them. If so, the client host will at least

know what has happened (because the reservation request carried in the *reopen* RPC will fail).

One (rather crude) approach to this problem is to set aside some disk space in anticipation of it. For example, some file systems, such as the Berkeley Fast File System [McKusick et al. 1984], reserve a certain amount of free space in order to obtain better performance. This free-space reserve, which can be used up by super-user processes on BSD systems, might also be employed to store blocks written back from embargoed clients. This solution, however, is at best a stopgap and can lead to some tricky management problems. For instance, what do you do when this space also runs out?

6.7. Congestion Avoidance during Recovery

The primary mechanism to avoid server congestion during recovery is the server-centric approach, in which the server tells clients to perform their recovery operations rather than helplessly accepting RPCs from all clients at once. The server may choose to do things entirely serially (that is, allow only one client at a time to do *reopen* RPCs), or it may send *reqreopen* RPCs to several clients in parallel. This latter approach will probably reduce the elapsed time of the recovery phase, although some care will be necessary to set the right level of parallelism. The server may be able to monitor its load during recovery and adjust the number of parallel operations accordingly. Because *reqreopen* sets a "credit limit" on the number of files a client is allowed to reopen, the server is able to react quickly when the load gets too high.

During the recovery phase, the server must not honor nonrecovery RPCs, or else inconsistencies might arise. In our implementation, the server simply drops such RPCs during recovery rather than sending an error reply. Not only does this protect vanilla-NFS clients from unexpected errors; it also reduces the load on the server, since clients will have to time out before retrying these RPCs.

The clients also try to reduce the server load during recovery, by suppressing the transmission of nonrecovery RPCs. They can do so by setting a flag in the appropriate data structure, causing the RPC transmission code to block on any RPC except *reopen*. When the client host receives the *endrecov* RPC, it unblocks any processes waiting to send RPCs to the server.

7. Resolving Embargoes

A server embargoes a Spritely NFS client because it believes that the client's state may have become inconsistent as the result of a communications failure and that

the client may not be aware of it. When a client attempts to use a server that has embargoed it, the server returns a specific error code for all RPCs except *clctl*. The client must then resolve the embargo before continuing to use the server. Embargo resolution means determining which open files might be inconsistent and reestablishing access to files for which consistency still exists.

Note that any scheme for embargo resolution that detects all true conflicts will also give false positives; that is, a client may decide that a conflict exists when it does not. This means that the Spritely NFS client code must inform applications (or users, or system managers) about all potential inconsistencies, and let some higher intelligence decide what to do. Any "silent" failures could cause unacceptable errors; the embargo-resolution algorithms are meant to err on the side of caution.

The mechanisms described in this section are not yet implemented, but they do not seem to involve much complexity.

7.1. Clearing an Embargo

When the client finds out that it has been embargoed by the server, it must first clear the embargo condition using the *clctl* RPC. A scenario exists, however, where a simple version of this mechanism could lead to an inconsistent state. Suppose that the server tries to contact the client, times out, and declares an embargo at time T_1. At time T_2, the server receives an RPC from the client and returns the error code saying "you are embargoed," which causes the client to send a *clctl* RPC to clear the embargo. Suppose now that a duplicate of this RPC is delayed in the network. The first copy reaches the server at time T_3, which clears the embargo, and then unsuccessfully tries to contact the client again. The server declares a new embargo at time T_4, and then the delayed duplicate *clctl* arrives, causing the server to believe that the client has cleared the second embargo. The client, however, does not realize that a second embargo has been declared and could end up with an inconsistent cache.

We could solve this problem by doing a three-way handshake between the client and server, but instead we use a method based on synchronized clocks. It is safe to assume that the clocks are synchronized to within a few seconds. This is necessary anyway for proper operation of NFS, and is more than possible using a protocol such as NTP [Mills 1991], which normally achieves clock skews on the order of milliseconds.

In this approach, the server records (in volatile storage) the time at which a client is declared as embargoed. The client passes its current time in the *clctl* RPC; only if the RPC was issued after the embargo was declared will the server

accept it. If the client's clock is slower than the server's, it can keep retransmitting the RPC (with a new timestamp) until the server accepts it.

If the client's clock is much faster than the server's, this method could still fail. The condition for correctness in the worst case is that the clock skew must be less than the amount of time it takes the server to declare an embargo. Since this is on the order of several seconds, with reasonable clocks, it should not be a problem.

7.2. *Detecting Potential Inconsistencies*

The client needs an algorithm for deciding if the server granted conflicting access to an open file while the client was embargoed. A conflicting access is any write-access to a file open only for reading, or any access to a file for which the client has cached dirty data. Of course, we do not want the conflict-detection scheme to give more false positives than necessary.

There are two different ways a client can become embargoed by a server: a *callback* RPC to the client could fail, or the client could fail to respond during recovery. In the first case, the server explicitly knows that the embargoed client may have an inconsistency with respect to a specific file. In the second case, the server has no knowledge that the client is using a specific file, which forces us to find an algorithm that relies only on information available to the client.

What the client needs to know is whether a conflicting access has been made to the file following the client's last successful access. As it happens, the NFS protocol already provides this information. The standard NFS *read* and *write* RPCs always return the current attributes of a file, and the returned attributes include the last-access and last-modification timestamps (*atime* and *mtime*, respectively).

To detect a potential conflict, the embargoed client must obtain the current attributes from the server and perform the appropriate comparison. The timestamps offer at least one-second resolution. Therefore, to prevent the client from drawing the wrong conclusion, the server must wait at least one second before declaring an embargo (we would hardly expect the RPC timeout period to be shorter than that).

The server must also ensure that updates to the timestamp attributes survive across crashes. Otherwise, the following scenario will lead to an undetected inconsistency:

1. Client A, with exclusive write access to a file, caches dirty data.
2. The server crashes and recovers, failing to contact client A.
3. Client B opens, reads, and closes the file.
4. The server crashes before *atime* is updated on stable storage.

5. The server recovers, and client A realizes that it is embargoed. It samples the *atime* of the file, decides that no conflicting access has been granted, and writes its cached data back to the file.

Unfortunately, Client B's view of the file may not be consistent with the dirty data that client A writes.

We would rather not insist that the server update *atime* on stable storage before replying to every *read* RPC; it would make read access much slower. This is not necessary; the server need only stably update *atime* in two situations:

1. on the first read access to a file after embargoing a client using that file
2. on the first read access to any file after rebooting, if any client was embargoed during recovery.

Since embargo declarations are rare, we can simplify the first situation to be "the first read accesses to all files after any client has been embargoed." Then the two cases may be combined into a simple mechanism: the server keeps a global timestamp reflecting the most recent reboot or embargo declaration. If, on a *read* RPC, the initial access time of a file is less than this timestamp, the attributes must be made stable before responding to the RPC.

As for the *mtime* attribute, the server normally updates its value whenever a *write* RPC causes any change to the amount of disk space allocated to the file. Still, there are times when *mtime* is not necessarily updated immediately (e.g., with writes that do not change the file size). The conflict-detection algorithm requires that *mtime* be made stable on the first write access to a file after a client using that file is embargoed, or after rebooting if any client was embargoed in recovery. In other words, we can use a policy analogous to that used for the *atime* attribute: if, on a *write* RPC, the initial modify-time of a file is less than the global timestamp, the attributes must be made stable before responding to the RPC.

Note that the file version number returned by the Spritely NFS *open* RPC is not sufficient to detect possible conflicts, because it is not updated on read-only accesses to a file.

7.3. A Resolution Algorithm

When a client learns that it has been embargoed by a server, it goes through the following sequence:

1. Mark, in its table of active files, all open and closed-dirty files on that server as needing embargo resolution.

2. Internally inhibit any RPCs to the server, except for those issued by this resolution algorithm.

3. Perform a *clctl* RPC on the server, with an op-code that means "clear my embargo"; the server then will let the client perform normal RPCs.

4. Run through the set of files needing embargo resolution, performing an *open* RPC on each one. Two consequences follow:
   a. If the server has crashed and recovered, it now knows that the client wants access to the files. For closed-dirty files, the client should open them as if for write access.
   b. The client obtains the current attributes for each of the files.

5. As the current attributes for the files are obtained, the client compares the *atime* and *mtime* values against its cached copies of the attributes. If a conflicting access has occurred, the file is put into an error state.

6. When checking what was a closed-dirty file, which is now open for writing:
   a. If no conflicting access has occurred, put the file back into the closed-dirty state, using the *close* RPC to reserve sufficient disk space.
   b. If a conflict was detected, close the file without reserving any space.

7. After all files have been checked, reallow all RPCs to the server.

If the server reembargoes the client during this sequence, it can be restarted safely from scratch when the network partition heals.

7.4. Repairing Inconsistent Files

After the embargo resolution algorithm has been run, some of the client's files might be marked as being in an inconsistent state. (If the embargo occurred because a space reservation expired, it may be impossible to write the modified data back to the server). I do not pretend to know how to solve this problem, and in general it cannot be solved without some knowledge of the particular application. The LOCUS distributed system [Popek et al. 1981] knew how to resolve partitioned updates to directories and mailbox files. The "reintegration" techniques used in the Coda system [Satyanarayanan et al. 1990] might also prove useful. The best solution is to build robust networks so that partitions are rare and the problem seldom arises.

8. Performance

Our original goal with Spritely NFS was to improve performance over NFS. Since NFS does not need to support a recovery protocol, we must show that the added recovery overhead in Spritely NFS does not eliminate our advantage. Note that the original, nonrecovering version of Spritely NFS did better than NFS on realistic benchmarks even though NFS does not have to do any *open* and *close* RPCs; that is, Spritely NFS saves enough through better use of the client cache to make up for the extra RPCs.

The recovery protocol has two kinds of costs: in normal operation, a small overhead on each RPC and after a server crash, a recovery phase. Since NFS has no recovery phase, it will always be faster at continuing after a server reboot. These events should be rare, so the cost of recovery will be amortized over a long period of useful work. At any rate, the server-centric approach should allow us to do efficient recovery, since we are not put at risk of server overload during the recovery phase.

8.1. Performance of Spritely NFS in Normal Operation

In the original paper on Spritely NFS [Srinivasan and Mogul 1989], we presented some measurements showing that Spritely NFS outperformed NFS, even though it may have to do extra RPCs (*open* and *close*). Spritely NFS, through its use of delayed writes, achieved more parallelism (and sometimes fewer total *write* RPCs) than NFS. In the intervening years, the state of the art in NFS implementation has improved considerably. The most important changes have reduced the cost of NFS *write* operations, and so one might expect the relative advantage of Spritely NFS to drop. In many cases Spritely NFS also uses fewer *read* and *getattr* RPCs. The use of faster disks, faster server CPUs, and bigger file caches should also reduce this relative advantage.

There are few good benchmarks of overall file system performance. (The LADDIS [Keith and Wittle 1993] benchmark is not appropriate, because it specifically tests the performance of an NFS server and not that of the entire client-server system.) Our original paper used the "Modified Andrew Benchmark" [Ousterhout 1990], a single-client benchmark that simulates a software development task but produces numbers that are independent of the host's native compiler. This paper uses the same benchmark, although it is not necessarily representative of typical applications. I modified it slightly to write back all dirty blocks before timing is started (that is, to start with a clean buffer cache) and to execute certain

frequent commands (`grep`, `wc`) from the local `/tmp` directory rather than via NFS.

I ran the benchmark on various combinations of three different configurations:

Slow: DECstation-3100 (approximately 11.3 SPECmarks), 24 Mbyte RAM, RZ23 disk

Medium: DECstation-5000/200 (approximately 18.5 SPECmarks), 48 Mbyte RAM, RZ58 disk

Fast: DECstation-5000/200 (approximately 18.5 SPECmarks), 48 Mbyte RAM, RZ58 disk, NVRAM.

The "Fast" system is in fact the same system as the "Medium" system, except that it also has PrestoServe nonvolatile RAM. This means that, when it is acting as a server, NFS writes do not need to be synchronous with the disk. The systems were connected over an Ethernet, which bore little additional traffic during the benchmark trials.

Both systems ran identical software and could act as either client or server. The ULTRIX 4.3 NFS server implementation supports the "write-gathering" technique, which allows several *write* RPCs in a row to be satisfied while only performing one update of the file system's overhead data.[4] Identical kernels were used in all tests; a flag variable was used to enable or disable Spritely NFS behavior. Another flag controls whether local-disk writes are delayed; although by default in ULTRIX this flag is cleared, during these benchmark trials delayed writes were enabled.

Each system provided four NFS server threads and four "block–I/O daemon" threads; these are default values. Folklore suggests that using more threads improves performance (especially with the write-gathering technique), but I measured no significant performance change when using 20 of each kind of thread, probably because the benchmark uses relatively small files, and so extra parallelism in the form of asynchronous writes is not available.

The benchmark has five phases:

1. directory creation
2. file copying
3. recursive directory operations
4. scanning of all files
5. compilation and linking.

4. This technique appears to have been independently invented by several people [Juszczak 1994]. Epoch's HyperWrite was apparently the first such product to ship [Bowen 1990].

Very little user-mode CPU time is expended during the first four phases. The last phase, compilation, does require substantial user-mode computation: approximately 71 seconds on the Slow system, and 42 seconds on the Medium/Fast system.

Table 2. Performance on Modified Andrew Benchmark. Numbers are time in seconds, averaged over five trials; differences of less than one second are not significant.

Client	Server	Type	Update policy	Phase 1 mkdir	Phase 2 copy	Phase 3 ls -R	Phase 4 find	Phase 5 make	Total time
Fast		Local	PU	0.2	3	6	7	57	73
Fast		Local	IPU	0.4	3	7	6	55	70
Medium		Local	PU	2	5	7	7	67	88
Medium		Local	IPU	2	5	6	6	64	83
Slow		Local	PU	4	12	15	14	126	171
Slow		Local	IPU	4	11	14	11	117	158
Fast	Slow	NFS	PU	3	21	7	6	97	134
Fast	Slow	NFS	IPU	4	21	7	6	96	134
Fast	Slow	SNFS	PU	4	15	8	8	76	109
Fast	Slow	SNFS	IPU	4	14	7	7	74	105
Slow	Fast	NFS	PU	1	8	17	14	114	154
Slow	Fast	NFS	IPU	1	9	17	14	113	154
Slow	Fast	SNFS	PU	1	6	18	15	111	151
Slow	Fast	SNFS	IPU	1	6	15	14	111	148
Slow	Medium	NFS	PU	3	13	17	15	128	176
Slow	Medium	NFS	IPU	3	13	17	14	123	170
Slow	Medium	SNFS	PU	2	10	16	15	118	161
Slow	Medium	SNFS	IPU	3	9	15	14	114	156

Table 2 shows the results for various combinations of client and server and for two different update policies (see Section 3.3). Times are given in seconds and averaged over five trials for each configuration. From the last column of the table, one can see that the Individual Periodic Update (IPU) policy is a clear improvement over the Periodic Update (PU) policy, especially when the disk is slow (the RZ23 disk has a specified average access time of 26.8 msec, and the RZ58, 18.1 msec). Only in the case of NFS, where few writes are delayed, is the update policy mostly irrelevant. (The IPU policy does speed some of the NFS-based configurations somewhat, but only because the compilation phase writes considerable data to `/tmp`, which is a local disk in all cases.)

The relative performance advantage of Spritely NFS, based on the overall elapsed time for the benchmark, is summarized in Table 3. (These ratios are calculated from the results of the trials using the IPU policy.) With a slow server, Spritely NFS provides a distinct performance improvement (about 26% on the overall elapsed time). With a fast server, the improvement is smaller (about 4% overall). In practice, server speed is not solely a function of server hardware; a load placed on the server by other clients should also increase the relative advantage of Spritely NFS.

Table 3. Elapsed Time Ratios on Modified Andrew Benchmark.

Client	Server	NFS Total Time	SNFS Total Time	Ratio
Fast	Slow	134 sec	105 sec	1.26
Slow	Medium	170 sec	156 sec	1.09
Slow	Fast	154 sec	148 sec	1.04

These trials were run with the `/tmp` directory on the client's local disk, which seems to be a realistic configuration now that people are less enamored of fully diskless workstations. Since the compilation phase does a number of write-read-remove sequences on temporary files, one might expect Spritely NFS to show an even larger advantage on diskless clients.

One can also gain some insight into the effects of the cache-consistency protocol by counting the number of different RPC operations performed, as shown in Table 4. The table shows the mean RPC counts for the configuration that gained the most from Spritely NFS (Fast client, Slow server, and the IPU update policy). The RPC counts for the nominally local case are also shown, because I was unable to entirely prevent the client from doing a few remote operations. The counts in this column are fairly small; investigation shows that these background RPCs are due to the use of the NFS automounter [Callaghan and Lyon 1989] and do not actually go over the network.

Spritely NFS uses fewer *getattr*, *read*, and *write* RPCs than NFS does, at the expense of a number of *open* and *close* RPCs. The total number of RPCs used by Spritely NFS is somewhat lower, but the real performance advantage comes from write-behind and from more frequent client cache hits. Many of the reads done by NFS actually try to retrieve data beyond the current end-of-file; Spritely NFS can avoid these reads because, while a file is open and cachable, the client reliably knows the length of the file.

Table 4. RPC Operation Counts for Modified Andrew Benchmark.

RPC type	Local Disk	NFS	Spritely NFS
getattr	307	1,225	259
setattr	0	22	22
lookup	62	810	535
readlink	126	0	0
read	0	1,186	836
write	0	476	192
create	0	96	96
remove	0	6	6
rename	0	4	4
mkdir	0	20	20
readdir	0	157	157
open			703
close			764
callback			0
Total	495	4,002	3,595

Spritely NFS, in this benchmark, performs almost as many *lookup* and *getattr* operations as *read* and *write* operations. It thus seems likely that improving the caching of directory entries and file attributes could significantly improve performance; see Section 10 for some thoughts on how it might be accomplished.

8.2. Cost of Client List Maintenance

The additional per-RPC overhead in Spritely NFS comes from the maintenance of the client list. I argued earlier that this overhead is negligible; most of the time, we simply do a hash-table lookup to discover that the client is already known and not embargoed.[5] Very rarely, we must update stable storage, but it is unlikely that a server would see such a high rate of new clients that this overhead becomes measurable.

In a simple test (repeated invocations of the NFS *null* RPC, using a fast server), the client list check added an average of about 28 microseconds, or about

5. The per-RPC operations in the original Sprite recovery mechanism apparently made a small but measurable difference in the RPC overhead, perhaps because on each RPC request and reply, the code was forced to manipulate timers.

2.3 percent. On actual RPC requests, the server performs far more work, so the overhead ratio would be even lower. The per-RPC overhead for client-list checks does not cause a measurable difference in Spritely NFS performance on realistic benchmarks.

8.3. Cost of Server Recovery

The cost of server recovery is essentially proportional to the number of files reopened in step II, since there are likely to be many more open files than active clients. When a client reopens N files, the number of *reqreopen* and *reopen* RPCs used is about $2 \times \lceil N/M \rceil$, where M is the number of files that can be described in one *reopen* request. M could be as high as 96, but the current implementation uses $M = 4$ to facilitate debugging. The server may have to go to the disk for each file reopened, so there may be $O(N)$ server disk accesses during recovery. In practice, the number of disk accesses could be much lower, since caching and spatial locality often obviate the need to read the disk each time a file is opened.

To get an idea of how long recovery might take, I induced a client host to keep 393 files open at once, did enough activity on the server to flush out its disk cache, and then triggered the recovery protocol. The complete recovery procedure took about two seconds of elapsed time, suggesting that recovery takes about 5–7 msec., on average, for each open file, but we cannot safely extrapolate from this simple measurement to predict how long it might take to recover a heavily used server.

8.4. Space Overhead

One disadvantage of "stateful" servers is that they require storage space to keep track of client state. A Spritely NFS server needs to keep a state-table entry for each file opened (or closed-dirty) by any client. The size of the table entry is proportional to the number of clients who have the file open. Thus, in the worst case, the state table requires storage proportional to the sum of the size of the "file tables" on all of the possible clients.

In our implementation of the server, state table entries consist of a fixed-length record describing the file state, possibly pointing to a linked list of smaller records recording client references to the file. Each of the file-state records uses 72 bytes; each of the client-state records uses 20 bytes. With some clever encoding, the file-state record could be shrunk to perhaps 48 bytes and the client-state record could be shrunk to 16 bytes, but even so the space requirements are not insignificant.

For example, suppose that a server has 100 clients, each of which has 1,000 different files open. This would require the server to maintain 100,000 records

of each type, using up more than 9 megabytes of kernel memory (or 7 Mbytes if better encodings are used). Although memory densities and costs improve rapidly from year to year, that is not an inconsequential amount of storage. Also, after a server crash it must all be reconstructed by the recovery protocol, which might take some time.

Nor is this necessarily a worst-case scenario. If clients are allowed to cache attributes of files that they do not currently have open (see Section 10), the potential size of the state table could be proportional to the number of files in the server's file system multiplied by the number of clients.

John Ousterhout [Ousterhout, personal communication, 1992] has suggested that if the server runs out of space in its state table, it could select certain entries (perhaps using an LRU scheme) to be discarded. Before discarding an entry, the server would inform the relevant clients via a *callback* RPC. Clients with cached dirty data would have to write it back to the server at this point. Once a client has been told that a file's entry has been discarded, it could not cache or access that file before reopening it; this means that no potential cache inconsistency is introduced.

This mechanism would work like a demand-paged memory system, except that in this case the "backing store" consists of the clients' file tables rather than stable storage. If the server crashes, during recovery the clients would not have to reopen these "paged-out" entries, and so the duration of the recovery phase might be shortened. Of course, just as with demand-paged memory, if the working set is larger than the available storage, the system will thrash.

9. *Software Complexity*

Since I have described this recovery protocol as "simple," it seems appropriate to describe how much work it would take to convert an NFS implementation to support Spritely NFS with recovery. The original Spritely NFS implementation was written in the course of a month or so by a programmer who had never before studied the UNIX kernel. A prototype implementation of the recovery protocol also took about a month. Several more months were spent cleaning up these prototype implementations, revising the protocol, and finishing off loose ends.

This section sketches the existing Spritely NFS implementation. Space does not permit the discussion of many tricky details, which are present in any distributed file system implementation.

9.1. Client Implementation Overview

Starting with the ULTRIX 4.3 client NFS implementation, the modifications necessary to support Spritely NFS are fairly straightforward. The *open* and *close* operations have to be implemented, the per-file data structures need to include cachability information, and the data access paths need to observe the cachability information. The most complexity by far comes from dealing with write-behind, since this involves a number of asynchronous operations and violates assumptions made all over the original NFS code.

The client needs to run a daemon process to handle callback requests. It turns out that the existing NFS server daemon does this just fine; the NFS server code in the client kernel handles the new callback RPCs. Some care has to be taken to avoid tying up all the daemon threads handling callbacks, or else two Spritely NFS systems serving each other might deadlock.

Very few changes are needed in other components of the client operating system. The code that manages the table of open and closed files (the equivalent of the UNIX *inode* table) must inform the Spritely NFS client code when a closed-dirty file is being removed from this table to make room for a new entry. The space-reservation mechanism requires a mechanism to count the amount of cached dirty data. It is also useful to provide a mechanism to remove dirty blocks from the file cache, for use when the file that contains those blocks is deleted; this improves performance by eliminating useless write-backs. We also had to fix a few bugs that were only tickled by Spritely NFS.

9.2. Server Implementation Overview

The changes to the server are in some ways simpler than the client changes, although they involve more new lines of code. For Spritely NFS without recovery, the changes were quite localized: the existing RPC server procedures were not touched, and all the new code related to handling the *open* and *close* RPCs and performing callbacks.

To support recovery, the server code that dispatches RPC operations must check (and, rarely, update) the client list on each RPC. Almost all of the remaining recovery protocol is implemented in a user-mode daemon program, which provides two services:

- When the kernel wants to update the stable copy of the client list, the daemon transfers the information to a simple database it maintains in a disk file.
- All of the RPCs generated by the server during the recovery phase (that is,

beginrecov, *reqreopen*, and *endrecov*) are issued by the daemon process. During recovery, the server's kernel code handles the *reopen* RPCs.

These two functions are done in separate processes to simplify the concurrency issues.

To allow these functions to be done outside the kernel, the kernel code does have to provide an interface by which it communicates with the user-level daemon. This is done using a number of *ioctl* commands. One of these passes client list modifications to the daemon process. The rest are used to control the kernel's internal client list database, to obtain the current recovery epoch (for use in the *beginrecov* and *endrecov* RPCs), and to suppress handling of nonrecovery RPCs during the recovery phase.

The space-reservation mechanism adds some code to the handler for the *write* RPC. It also requires some support from the underlying local file system, about 40 lines of new code, to do the necessary bookkeeping.

In order to provide full consistency between Spritely NFS clients and local file system applications running on the server, there would have to be some additional linkages between the local file system's *open* and *close* operations and the Spritely NFS state-table mechanism. For example, the opening of a file by a local process might require Spritely NFS to change a client's cachability information for that file.

9.3. Code Complexity Metrics

There are many ways to measure the complexity of a software system, but I will rely on just two: the number of lines of source code and the number of instruction bytes. By either measure, Spritely NFS is about 50 percent more complex than vanilla NFS.

The original ULTRIX 4.3 implementation of NFS comprises 12,283 lines in 14 kernel source files (not counting the RPC layer). This implementation of Spritely NFS comprises 19,419 source lines in 22 source files, for a net increase of 7,136 lines, or 58 percent. (Source line counts include comments and blank lines.)

The ULTRIX NFS code compiles into 8 object modules, containing 75,472 "text" bytes. The Spritely NFS code compiles into 14 modules containing 119,264 text bytes, also for a net increase of 58 percent.

The current Spritely NFS code is actually larger than it might be in a final implementation, because it is somewhat more heavily commented than the NFS code, it includes blocks of the original NFS code "#ifdef'ed out," and it includes

numerous debugging statements. On the other hand, several features left to be implemented will also affect the code size.

The modules added for Spritely NFS are

Client callback/recovery code: handlers to serve the *callback* RPC and recovery-related RPCs (827 lines)

Server client-list and daemon interface: code to maintain client list in kernel and to interface to user-level, client-list/recovery daemon (696+90 lines)

New server RPCs, space reservation, and callback: handlers for the *open*, *close*, *reopen*, and *clctl* RPCs, support for space reservation, and code to perform consistency callbacks (940 lines)

Miscellaneous subroutines: for both client and server code (344 lines)

State table: code to maintain the server's state table database (920+96 lines)

State transitions: code to handle *open* and *close* state transitions (847 lines).

The bulk of the new modules is concentrated in server code; there are 3,589 lines of new server-side modules versus just 827 lines in one new client-side module (and 344 lines in one shared module). However, the changes made to existing modules are mostly to client code; 1,121 lines were added to client modules, 343 lines to server modules, and 912 lines to shared modules.

The user-mode daemon process, which provides stable storage for the client list and manages the recovery procedure, is about 1,300 lines of source code. This code is not particularly complicated, and the main code of the recovery protocol itself requires less than 200 lines. This implementation of the daemon does recovery one client at a time. To support parallelism in the recovery procedure, the daemon would have to manage multiple threads and so might be considerably more complicated.

9.4. Pitfalls

The fact that most of the changes to existing code were in client-side modules confirms our experience that the conceptual complexity is mostly in client code. In particular, it turned out to be quite tricky to add write-after-close to the existing NFS implementation, since the NFS code makes many assumptions about the quiescence of a file after it is closed. Bugs resulting from violations of these assumptions proved to be the most difficult to discover and repair.

The ULTRIX kernel supports symmetric multiprocessing and so uses explicit locks to protect kernel data structures. The existing locking design does not support some of the situations arising in Spritely NFS very well, and there are still a few races in the code that should be protected by new locks.

For example, when the client receives a *beginrecov* RPC, it must go through the table of open-file descriptors or "gnodes," looking for gnodes that must be reopened. During the process of examining one of these gnodes, we would like to be able to lock it, because some other process may decide to trash it while we are looking at it (which might cause us to dereference a dangling pointer and crash). However, we cannot use the normal gnode locking mechanism, because the gnode might already be locked by another process; that process may be blocked waiting for the server to finish recovery, and so if we tried to acquire the gnode lock at this point we would probably deadlock.

Obvious solutions exist that involve the use of additional locks, but these solutions add extra locking overhead to all file system operations. It would be nice to find a solution that did not have much extra cost except during recovery.

10. Future Extensions

Spritely NFS currently provides explicit consistency only for file data and for the attributes of currently open files. The protocol could be extended to provide true consistency for directories, for attributes of nonopen files, and perhaps for file locks. Although these extensions have not yet been implemented, they do have implications for the recovery mechanism.

For additional discussion of directory caching and attributes caching for Spritely NFS, see [Mogul 1992].

10.1. Directory Caching

A large fraction of NFS traffic consists of directory lookups and listings. Many NFS implementations cache directory entries, but because NFS has no consistency protocol these caches must time out quickly and even so can become inconsistent.

Recent measurements on Sprite suggest that it is better to cache (and invalidate) entire directories than to cache individual entries, since a directory is often the region of exploitable locality of reference [Shirriff and Ousterhout 1992]. In this regard, Sprite nicely matches the Spritely NFS model; the client simply does an *open* on a directory before doing *readdir* RPCs, and keeps the result of the *readdir* in a cache. When the client removes a directory from its cache, it does a *close* RPC to inform the server. If a different client modifies the directory (using an RPC such as *create*, *remove*, *rename*, etc.), then the server does a *callback* to cause the first client to invalidate its cache.

Whole-directory caching is especially effective at satisfying lookups for nonexistent entries. It can significantly improve performance because some applications (particularly compilers and command shells) often lookup names that do not exist, as they follow a "search path" looking for a file. Such "negative" caching requires guaranteed consistency in order to provide safe results for distributed applications.

Should clients write-through directory changes (i.e., creations, renames, or removals), or could changes be done using write-back? Write-back is far more complex, especially because it makes it much harder to provide the failure-atomicity guarantee that UNIX has traditionally attempted for directory operations. If only write-through is allowed, then *open* on a directory always allows the client to cache; it serves solely to inform the server of which clients might need callbacks when an entry is changed.

Moreover, during server recovery the clients can simply flush their directory caches. When, subsequent to recovery, a client needs to read a directory, it then does a fresh *open* to notify the server that it wants to see callbacks. Reloading the directory caches, rather than revalidating them (as is done with cached data), is unlikely to be expensive because even a small directory cache yields a high hit rate [Shirriff and Ousterhout 1992].

Write-through also avoids having to coordinate the allocation of file identifiers between the client and the server when new files are created. The client transmits the *create* operation to the server, which allocates a file identifier, inserts a new directory entry, and returns the new identifier to the client. The client can then update its cached copy of the directory. A write-back scheme for file creation would be much more complex.

10.2. Attributes Caching

Spritely NFS provides consistency for file attributes (length, protection, modification time, etc.) only for open files. Some applications (for example `make`, `ls -l`, and `du`) use the attributes of files that they will not (or cannot) open. Because attributes are read so often, NFS implementations are forced to provide "attribute caching" using a probabilistic consistency mechanism: cached attributes time out after a few seconds.

Spritely NFS could be extended to support consistent caching of attributes, using some sort of open-for-attributes-read RPC. This RPC, which might be called *openattr*, would return the current attributes, thereby avoiding the need for a *getattr* RPC, and would also tell the client if the attributes were cachable or not. The *lookup*, and optionally the *close*, operations could also indicate that the client wants to cache a file's attributes. The server would then use a *callback* to

invalidate a client's cache entry when some other client changes the attributes of the corresponding file.

It is not clear that consistent attributes caching is necessary; experience with NFS has shown that weak consistency is usually sufficient, because few applications depend on strong consistency for unopened files. (Weak attribute consistency on open files causes errors when two client hosts simultaneously attempt to append to the same file, because they have an inconsistent view of the length of the file.)

It is unclear if introducing consistent attributes caching will help or hurt performance. It might obviate many of the *getattr* calls now done by NFS (see Section 8.1), but it also might greatly increase memory requirements at the server, for tracking the state of cached attributes (see Section 8.4).

If consistent attributes caching is added to Spritely NFS, clients would presumably still write-through all attributes changes.[6] It would not be possible to use a write-back policy for attributes changes without making widespread changes to the client NFS implementation, and it probably would not be worth the effort (since explicit attribute modifications are rare).

As with directory caches, use of write-through for attributes means that during server recovery the clients only need flush their caches of attribute information for nonopen files. After recovery, a client wishing to cache a file's attributes would again do an *openattr* RPC on the file. It would not be cost effective to do a *reopen* during recovery for every attribute-cache entry because the cost of the *reopen* would be no less than the cost of simply doing the *openattr*. Doing the work after recovery shortens the recovery phase and avoids reloading cached information that might not be needed.

10.3. File Locking

Although write-sharing is normally rare, when processes do write-share a file, they often use a locking mechanism to serialize access to the file. Because the basic NFS protocol does not provide file-locking primitives, a separate "NFS locking protocol" is often used [X/Open Company 1992]. This protocol is typically implemented in user-level processes on both the client and server. When a client application issues a lock command, the client kernel forwards the lock operation to the lock daemon process on the server via the local lock daemon process. The client kernel also marks the file as uncachable, in order to avoid consistency

6. Except for file-length changes caused by writes, which would destroy the performance advantage of write-behind. A client would not be allowed to cache attributes of a file currently open for writing (or closed-dirty) by another client.

problems. An associated status protocol detects host reboots and causes the lock daemons to resubmit their lock requests.

For Spritely NFS, with its explicit consistency protocol and recovery mechanism, it would make sense to design a locking mechanism that is part of the main protocol. This was not done for NFS presumably because it would have introduced server state, but it would solve a number of performance problems with the current locking mechanism:

- When NFS clients do lock operations, the lock server must participate because the clients have no way of knowing if other client hosts are involved. In Spritely NFS, if a client host has the right to write-cache a file, then it need not contact the server to do locking, since no other client host could be using the file. When the file becomes shared and the server does a *callback*, the client would then forward its lock status to the server.
- NFS clients disable data caching when using locking, because this strongly implies that another client may be using the file and their caches might be inconsistent. Spritely NFS clients can continue to cache as long as no write-sharing is taking place.
- Because NFS locking is implemented outside of the kernel, every lock operation causes at least a pair of context switches on the client host, and several extra domain-crossings (between kernel and user) on the server. This extra overhead can make locking quite costly. If Spritely NFS provided locking as part of the main protocol, it would be implemented using the kernel-to-kernel RPC path and would not require extra domain crossing or context switching.

Spritely NFS would not need a separate protocol for recovering client lock status after a server crash. During the second step of the recovery phase, clients would be told to resubmit their lock requests to the server after they have reopened their files. Unlike the NFS status protocol, in which clients rely on timeouts to discover server recovery in time to rebuild their locks, the Spritely NFS recovery protocol guarantees that clients will be able to rebuild locks before recovery completes.

In UNIX, writes to regular files are atomic: even if a given write system call involves multiple disk blocks, if another process issues a single read system call for a subset of the region being written, that read will return either only old data or only modified data. NFS cannot make this guarantee, because the server cannot lock a file against reads during a multiblock write. Spritely NFS, in its current form, also does not directly provide atomicity. For both NFS and Spritely NFS, atomicity could be provided using a locking mechanism. Spritely NFS has the

advantage that the extra locking overhead is necessary only if write-sharing is actually taking place.

10.4. Security

NFS does not provide much in the way of security, but in principle one can use cryptographic techniques to prevent illicit access to file data [Kazar 1988; Taylor and Goldberg 1986]. Spritely NFS introduces the possibility of malicious interference with the cache-consistency and recovery protocols. Fortunately, the worst that could be done would be to slow down legitimate clients (perhaps by forcing them to stop caching the files they are using, or wait for spurious *callbacks* to time out).

One probably could add authentication mechanisms to the Spritely NFS protocol to prevent an intruder from interfering with access to a file that it was not otherwise authorized to use, and to reject spurious callbacks. But because they would not eliminate the threat of denial-of-service attacks and could add considerable overhead to the protocol, they are probably not worth the effort.

10.5. Memory-Mapped Files

Many operating systems support memory-mapped files: an entire file appears in the region of a process address space, and file reads and writes occur as the result of virtual-memory operations instead of explicit system calls. Memory-mapped files are a convenient way to implement distributed shared memory, if an efficient consistency mechanism is available. NFS does not provide sufficient consistency for shared memory applications. Sprite (as well as Spritely NFS) provides consistency but not efficiently, because when a file is write-shared, Sprite forces the clients to contact the server on every access. That is, Sprite does not allow a distributed-shared-memory application to use page-level caching to take advantage of locality of reference.

The Mether-NFS system [Minnich 1993] uses the standard set of NFS remote procedures to implement a cache-consistent, distributed shared memory. Unlike Sprite, MNFS allows clients to cache pages from a write-shared file. The MNFS server keeps a map recording the current readers and writer of each page of the file. When a client wants to use a page for which it does not currently have appropriate access rights, it contacts the server (using an NFS *read* RPC), using some high-order bits in the file-offset argument to indicate if it wants to read or write the page. The server in turn contacts the current writer of that page and retrieves the current value, using an NFS *read* RPC as a sort of *callback*. The server may also cause any cached copies at other clients to be invalidated, if the client intends to write the page.

One could extend Spritely NFS to support the MNFS approach to write-shared files. This would require a recovery mechanism for the server's page-map. A simple way to accomplish this is to start by using the current recovery algorithm to rebuild the server's state-table entry for the write-shared files. Then, the server would *callback* each client using a write-shared file, forcing them to write-back all dirty pages for the file and to invalidate all cached pages for the file.

At this point, because no clients are caching pages from the file, the server can recreate its page-map for the file as an array of "empty" records, which is a consistent state for the page-map. This approach pages the entire mapped file out to the server's disk and then pages it back in as clients start to use it again, so it could be quite slow.

It would be more efficient to rebuild the server's page-map directly, using information held by the clients. In this approach, clients would not write-back or invalidate their shared-file pages. Instead, the server would *callback* to each client to obtain the status of the pages in a given file. The client would return a list of the pages it holds in its cache, indicating which of them are writable.

All MNFS files are backed by actual server disk files; because MNFS follows NFS semantics for server writes, any pages written back to the server would survive crashes. However, a dirty page held by a failed client would be lost, making it impractical for MNFS (or a similar extension to Spritely NFS) to protect applications from client crashes.

10.6. Preventing Modification of Busy Text Files

When a process is executing a program, UNIX normally prohibits modifications of the program text file. This prevents the executing process from seeing the inconsistencies that could arise if, for example, a page-fault resulted in reading a page from the modified file into an instruction stream made up of pages from the original file.

Because one NFS client cannot know that some other NFS client is currently executing from a file, this guarantee cannot be fully preserved in an NFS environment.

The current version of Spritely NFS is also unable to prevent modification of an active text file; in fact, by guaranteeing consistency, Spritely NFS might make things worse than NFS does. If an NFS client executing the program is lucky enough to have the entire text in its memory, it will not have to fault in any pages from the server. A Spritely NFS client with a "unified buffer cache" will immediately discover that its cache is invalid, and so will immediately corrupt its instruction stream.

The Spritely NFS protocol could be modified in at least three different ways to solve this problem:

1. The open and close RPCs, in addition to specifying the number of readers and writers, could also indicate if the client intends to execute the file. The server would return the ETXTBUSY error code in reply to any attempt to open an executing file for writing. Sprite uses this approach [Ousterhout, personal communication, 1992].
2. The client kernel could use the file-locking protocol to prevent modification of a file that it is executing. This, however, requires a mandatory locking protocol; many UNIX systems do not support mandatory locking.
3. The server could reject an attempt to open an executable file for write access if the file is already open for reading. This is slightly more restrictive than necessary, but is easy to implement.

11. Other Related Work

Several interesting papers related to recovery in distributed file systems have never been published. Rick Macklem worked on "Not Quite NFS," an attempt to use the leases model to provide recovery for an NFS extended with a Sprite-like consistency protocol [Macklem 1994]. Meanwhile, the Echo file system project at Digital's Systems Research Center has grappled with a number of similar issues, especially those related to write-behind [Mann et al. 1993].

Mary G. Baker and Mark Sullivan describe a similar approach to state recovery [Baker and Sullivan 1992], using a "recovery box": stable storage for selected pieces of system state, to allow a system to reboot quickly. In their approach, a file server would store all the open file handles in stable storage, with the assumption that these are unlikely to be corrupted by (or just prior to) a crash. Spritely NFS is more conservative, because it does not require a large chunk of low-latency stable storage, and it makes far weaker assumptions about the effects of a crash. Their system, however, leads to much quicker recovery.

12. Conclusion

Adding cache consistency to NFS was an interesting experiment, but without a recovery protocol Spritely NFS was not suitable for production use. The recovery mechanisms described in this paper should be enough to make Spritely NFS a real

alternative to NFS. The recovery mechanism is so simple, especially on the client side, that one can no longer claim that only a stateless protocol admits a simple implementation.

Even if Spritely NFS never becomes widely used, I believe that this simplified approach to recovery will be useful in other contexts. A similar approach is being used now in Sprite, and their experiences should help to validate the design.

Acknowledgments

The design in this paper has evolved (sometimes discontinuously) in lengthy exchanges among many people, including (in alphabetical order) Mary G. Baker, Cary Gray, Chet Juszczak, Rick Macklem, Larry McVoy, John Ousterhout, V. Srinivasan, Garret Swart, and Brent Welch. Most of these people have talked me out of at least one bad idea. V. Srinivasan and Bharat Shyam each contributed an entire summer to this project and were responsible for much of the implementation. Cary Gray, John Ousterhout, and the anonymous reviewers made numerous suggestions that improved the paper.

Appendix A: Protocol Specification

One should read the body of this paper before attempting to understand the following brief specification for the Spritely NFS protocol, which reflects the state of the Spritely NFS implementation at the time this paper was written. Because the actual protocol changes constantly, the specification should be viewed as a research prototype, not as a standard. The original NFS specification [Sun Microsystems 1989] is included as a subset and is not repeated here.

A.1. New Error Codes

The `stat` enumeration includes several new values:

```
NFSERR_TABLEFULL=100,
NFSERR_CALLBACKFAIL=101,
NFSERR_NOTRECOVERING=102,
NFSERR_EMBARGOED=103,
NFSERR_INCONSISTENT=104,
NFSERR_TRYLATER=105
```

The meanings of these values are:

NFSERR_TABLEFULL: The server's state table is full and no new files may be opened.

NFSERR_CALLBACKFAIL: An NFSPROC_CALLBACK RPC from a server to a client is rejected because it is malformed or out of sequence.

NFSERR_NOTRECOVERING: An NFSPROC_REOPEN RPC is rejected because the server is not currently in its crash-recovery phase.

NFSERR_EMBARGOED: The client is embargoed from further operations until it takes steps to clear the embargo. This code may be issued in response to any RPC, except for NFSPROC_CLCTL.

NFSERR_INCONSISTENT: Some violation of the cache-consistency protocol has led to a state that is either inconsistent, or would be inconsistent if the requested RPC were to be executed.

NFSERR_TRYLATER: An NFSPROC_CALLBACK or NFSPROC_OPEN RPC cannot be completed without deadlocking or blocking for a long interval. The RPC should be retried after a few seconds. The intent of this status code is to avoid having an unbounded delay on any RPC call.

A.2. New Data Types

The openargs data type is used with the NFSPROC_OPEN RPC (and indirectly with the NFSPROC_REOPEN RPC) to communicate the client's current open-for-read and open-for-write reference counts.

```
typedef struct {
     fhandle file;
     unsigned mdev;
     unsigned rcount;
     unsigned wcount;
} openargs;
```

For the specified file, the client's current count of read-only opens is rcount, and the current count of read-write opens is wcount. The mdev field is used to disambiguate a file handle during a callback. The server does not otherwise interpret the mdev field.

The openres data type is used in the reply to an NFSPROC_OPEN or NFSPROC_REOPEN RPC.

```
typedef union switch (stat status) {
   NFS_OK:
      struct {
         unsigned cacheVers;
         unsigned oldCacheVers;
         boolean cacheEnabled;
         fattr attributes;
      }
   default:
      struct {}
} openres;
```

The `cacheVers` field carries the current version number of the file, which changes on each open-for-write. The `oldCacheVers` field carries the previous version number. This allows a client to decide if its cached data from the file is still valid: if it matches the `cacheVers` field, then this file has not been opened for write since the data was cached; if it matches the `oldCacheVers` field, then the version changed because of the client's current open-for-write operation.

The `cacheEnabled` field indicates whether the client is allowed to cache the file.

The `attributes` field contains the current attributes values for the file. If the file is cachable, this allows the client to avoid doing a subsequent `NFSPROC_GETATTR`.

The `reopeninfo` data type is used with the `NFSPROC_REOPEN` RPC to pass information about a file that a client currently has open (or for which it has cached dirty blocks).

```
typedef struct {
      openargs openargs;
      unsigned dirtybytes;
} reopeninfo;
```

The `openargs` field contains the information needed by the server to rebuild its state-table entry for an open file. The `dirtybytes` value is necessary when rebuilding a state-table entry for a closed-dirty file; see the description of `NFSPROC_CLOSE` for more details.

A.3. Existing RPC Server Procedures

None of the RPC definitions in the original NFS specification [Sun Microsystems 1989] have been changed in Spritely NFS. The only exception is that a Spritely NFS server may return a status value of `NFSERR_EMBARGOED` in reply to any RPC request (but only if the client is known to be using Spritely NFS).

A.4. New RPC Server Procedures

These procedures are added to the NFS server to support Spritely NFS consistency and crash recovery.

```
18 NFSPROC_OPEN (file) returns (reply)
     openargs fileargs;
     openres reply;
```

This RPC is used to inform the server that the client is increasing one of its open-file reference counts for a specified file. It may not be used to decrease the reference counts. Because the client's current value of the reference count rather than the net change is transmitted, this RPC is idempotent.

```
19 NFSPROC_CLOSE (file, mode, rcount, wcount, dirtybytes)
                          returns (status)
     fhandle file;
     unsigned rcount;
     unsigned wcount;
     unsigned dirtybytes;
     stat status;
```

This RPC is used to inform the server that the client is decreasing one of its open-file reference counts for the `file`. The `rcount` argument carries the number of read-only references; the wcount argument carries the number of read-write references. The `dirtybytes` argument is used only when the reference counts are both zero, and the file had been both open for writing and cachable. The value of `dirtybytes` must be an upper bound on the number of bytes that the client has buffered for subsequent write-back for this file.

If the value of `dirtybytes` is non-zero, and the server replies with `NFS_OK`, the client is responsible for eventually issuing another `NFSPROC_CLOSE` RPC when all the dirty data has been written back. This final `NFSPROC_CLOSE` may be elided if the client opens the file again.

Note: this RPC should be modified to pass the current length of the file, in addition to the number of dirty bytes, in order to allow implementation of space-reservation by a server whose local file system does not support the concept of file holes. In such a file system, it is not possible to estimate the number of blocks that will be allocated, knowing only the number of bytes of data to be written, because a single delayed NFS write might extend the length of a file by several gigabytes. (A server may avoid implementing the space-reservation mechanism, at the expense of reduced performance. Such a server would always return NFSERR_NOSPC if dirtybytes is non-zero.)

If the server returns NFSERR_NOSPC in response to this RPC, the client should try to write all the buffered dirty data (using NFSPROC_WRITE), which might succeed. The client should then repeat the NFSPROC_CLOSE operation with dirtybytes equal to zero.

```
21 NFSPROC_REOPEN (epoch, files) returns (reply)
     unsigned epoch;
     reopeninfo files<REOPEN_MAXFILES>;
     union switch (stat status) {
        openres results<REOPEN_MAXFILES>;
     } reply;
```

This RPC is used only during server recovery and only in response to an NFSPROC_REQREOPEN RPC, described in section A.5. The client uses it to report the current reference-count state of any files that it has open on the server, and the number of dirty bytes (see the description of NFSPROC_CLOSE) for files that it does not have open but for which it still has cached dirty data. The files array carries one entry for each file to be reopened.

The server replies with an array of result values, corresponding to the files listed in the request RPC. Note that the server may change the cache-version numbering for a file after a crash, and so the client must update its view of the cache version as a result of this operation. Thus, cached data for files that the client does not reopen (i.e., closed, all-clean files) must be discarded during recovery, since the cacheVers returned by a subsequent NFSPROC_OPEN cannot be compared against the value held by the client.

The epoch argument is meant to carry the client's view of the server's recovery epoch (see NFSPROC_BEGINRECOV) to prevent the server from accepting requests from a previous recovery phase. This might be superfluous.

The value of REOPEN_MAXFILES is arbitrarily set to 96 to keep the size of the reply under 8k bytes. However, a client host normally should not try to reopen so many files at once, but should limit the size of each RPC and its reply so

that they fit into a single network packet. Also, the client must not reopen more files than specified by the `nfilesperreopen` argument provided in the server's `NFSPROC_REQREOPEN` RPC.

```
25 NFSPROC_CLCTL (flags, epoch) returns (status)
     unsigned flags;
     unsigned epoch;
     stat status;
```

This RPC is used by a Spritely NFS client when it reboots and when it is clearing an embargo. The *flags* argument is a set of bits:

001 Clear embargoed state

002 Client just rebooted.

The `epoch` value is used to make this RPC idempotent. When the RPC is used to report a reboot, the epoch must be higher than reported for any previous incarnation of the client; this informs the server that it can discard all of its state related to this client. The client should not retransmit this RPC with an increased epoch value, because its state will be lost.

When this RPC is used to clear an embargo, the client should set the epoch value to the current time, in seconds since midnight January 1, 1970 UTC. (This is the same format used for the `seconds` field of the `timeval` data type.) The server must then check that the value is greater than the time that the embargo was declared for this client; if it is not, the server should reply with `NFSERR_EMBARGOED`. The client should continue to retransmit with an updated `epoch` value until the server accepts the RPC.

A.5. New RPC Callback Procedures

These callback procedures are handled by the Spritely NFS client. That is, for these procedures the "RPC client" is the Spritely NFS server, and the "RPC server" is the Spritely NFS client.

```
20 NFSPROC_CALLBACK (file, code, mdev, attr)
                    returns (status)
     fhandle file;
     unsigned code;
     unsigned mdev;
```

```
    fattr attr;
    stat status;
```

This RPC informs the client that it no longer has permission to cache the `file` because some other client is now sharing it. The `code` argument is a set of bits, which may be ORed together:

001 write back dirty blocks

002 stop caching this file.

The `mdev` argument is the value that was passed by the client in its `NFSPROC_OPEN` RPC and is used by the client to locate its state related to the file. The `attr` argument provides the current attributes for the file.

```
23 NFSPROC_BEGINRECOV (epoch) returns (status)
    unsigned epoch;
    stat status;
```

This RPC is used to initiate recovery. The server sends it to each known Spritely NFS client. After receiving this RPC, the client should refrain from sending any RPCs (except `NFSPROC_REOPEN`, when prompted with an `NFSPROC_REQREOPEN`) until an `NFSPROC_ENDRECOV` is received. The client should also refrain from timing out pending RPCs.

When this RPC is received, the client should somehow record or mark the set of files that it has open (or closed-dirty) on this server so that it knows what needs to be reopened.

The client must reject this RPC if the `epoch` value is not greater than the `epoch` argument of the most recent `NFSPROC_ENDRECOV` RPC (if any have been received since the client booted).

```
24 NFSPROC_ENDRECOV (epoch) returns (status)
    unsigned epoch;
    stat status;
```

This RPC is used to conclude recovery. Once this RPC has been received, the client may resume normal use of the server.

The client must reject this RPC if the `epoch` value is not greater than the `epoch` argument of the most recent `NFSPROC_BEGINRECOV` RPC.

```
22 NFSPROC_REQREOPEN (epoch, nreopens, nfilesperreopen)
                          returns (reply)
     unsigned epoch;
     unsigned nreopens;
     unsigned nfilesperreopen;
     union switch (stat status) {
        boolean done;
     } reply;
```

This RPC is used only during server crash recovery. The server uses it to tell the client to reopen some of its files. The client may issue up to `nreopens` `NFSPROC_REOPEN` RPCs to the server, each specifying up to `nfilesperreopen` files. It then replies with the boolean value `done` set to TRUE if it has no more files left to reopen. If the client has not reopened all of its files, it replies with `done` set to FALSE, and the server issues another `NFSPROC_REOPEN` RPC.

The client must reject this RPC if the `epoch` value is not the same as the `epoch` argument of the most recent `NFSPROC_BEGINRECOV` RPC.

References

1. Baker, Mary G., John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed file System. In *Proceedings of the 13th Symposium on Operating Systems Principles*, 198–212. Pacific Grove, CA, October 1991.
2. Baker, Mary G. and John Ousterhout. Availability in the Sprite Distributed File System. *Operating Systems Review* 25(2):95–98, April 1991.
3. Baker, Mary G. and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment. In *Proceedings of the Summer 1992 USENIX Conference*,31–43. San Antonio, TX, June 1992.
4. Bowen, Ted Smalley. Software update speeds NFS write process on server. *Digital Review* 7(30):17, August 6, 1990.
5. Callaghan, Brent and Tom Lyon. The Automounter. In *Proceedings of the Winter 1989 USENIX Conference*, 43–51. San Diego, CA, February 1989.
6. Carson, Scott D. and Sanjeev Setia. Analysis of the Periodic Update Write Policy For Disk Cache. *IEEE Transactions on Software Engineering* 18(1):44–54, January 1992.
7. Gray, Cary G. and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles*, 202–210. Litchfield Park, AZ, December 1989.

8. Juszczak, Chet. Improving the Performance and Correctness of an NFS Server. In *Proceedings of the Winter 1989 USENIX Conference*, 53–63. San Diego, CA, February 1989.

9. Juszczak, Chet. Improving the Write Performance of an NFS Server. In *Proceedings of the Winter 1994 USENIX Conference*, 247–259. San Francisco, CA, January 1994.

10. Kazar, Michael Leon. Synchronization and Caching Issues in the Andrew File System. In *Proceedings of the Winter 1988 USENIX Conference*, 27–36. Dallas, TX, February, 1988.

11. Keith, Bruce E. and Mark Wittle. LADDIS: The Next Generation in NFS File Server Benchmarking. In *Proceedings of the Summer 1993 USENIX Conference*, 111–128. Cincinnati, OH, June 1993.

12. Macklem, Rick. Not Quite NFS, Soft Cache Consistency for NFS. In *Proceedings of the Winter 1994 USENIX Conference*, 261–278. San Francisco, CA, January 1994.

13. Mann, Timothy, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. *A Coherent Distributed File Cache with Directory Write-behind.* Research Report 103, Digital Equipment Corporation Systems Research Center, June 1993.

14. McKusick, Marshall K., William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems* 2(3):181–197, August 1984.

15. Mills, David L. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications* 39(10):1482–1493, October 1991.

16. Minnich, Ronald G. Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems* (SEDMS IV), 89–107, San Diego, CA, September 1993.

17. Mogul, Jeffrey C. A Recovery Protocol for Spritely NFS. In *Proceedings of the USENIX File Systems Workshop*, 93–109. Ann Arbor, MI, May 1992.

18. Nelson, Michael N., 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.

19. Ousterhout, John K. Why Aren't Operating Systems Getting Faster As Fast As Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, 247–256. Anaheim, CA, June 1990.

20. Ousterhout, John K., 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 *Proceedings of the 10th Symposium on Operating Systems Principles*, 15–24. Orcas Island, WA, December 1985.

21. Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th Symposium on Operating Systems Principles*, 169–177. Pacific Grove, CA, December 1981.

22. Rosenblum, Mendel and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th Symposium on Operating Systems Principles*, 1–15. Pacific Grove, CA, October 1991.
23. Satyanarayanan, Mahadev, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers* 39(4)447–459, April 1990.
24. Shirriff, Ken W. and John K. Ousterhout. A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System. In *Proceedings of the Winter 1992 USENIX Conference*, 315–331. San Francisco, CA, January 1992.
25. Srinivasan, V. and Jeffrey C. Mogul. Spritely NFS: Experiments with Cache-Consistency Protocols. In *Proceedings of the 12th Symposium on Operating Systems Principles*, 45–57. Litchfield Park, AZ, December 1989.
26. Sun Microsystems, Inc. *NFS: Network File System Protocol Specification.* RFC 1094, Network Information Center, SRI International, March 1989.
27. Taylor, Bradley and David Goldberg. Secure Networking in the Sun Environment. In *Proceedings of the Winter 1986 USENIX Conference*, 28–37. Atlanta, GA, June 1986.
28. Welch, Brent B. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System.* Ph.D. thesis, Department of Electrical Engineering and Computer Science. Technical Report UCB/CSD 90/567, University of California, Berkeley, February 1990.
29. X/Open Company, Ltd. *X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, Issue 4.* Reading, Berkshire, UK, 1992.