NFS Version 3
Design and Implementation

Brian Pawlowski
Chet Juszczak
Peter Staubach
Carl Smith
Diane Lebel
David Hitz

Abstract

This paper describes a new version of the Network File System (NFS) that supports access to files larger than 4GB and increases sequential write throughput seven-fold when compared to unaccelerated NFS Version 2. NFS Version 3 maintains the stateless server design and simple crash recovery of NFS Version 2, and the philosophy of building a distributed file service from cooperating protocols. We describe the protocol and its implementation, and provide initial performance measurements. We then describe the implementation effort. Finally, we contrast this work with other distributed file systems and discuss future revisions of NFS.

1. Introduction

"It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something." *Roosevelt, 1932*

The NFS protocol is a collection of remote procedures that allow a client to transparently access files stored on a server *[Joy84a]*. It is independent of architecture *[RFC1014]*, operating system, network, and transport protocol. The protocol does not exactly match the semantics of any existing system. Instead, it provides a basis for portability and interoperability.

NFS Version 1 existed only within Sun Microsystems and was never released. NFS Version 2 was implemented in 1984 and released with SunOS 2.0, in 1985 *[Sandberg85]*. NFS Version 2 implementations exist for a variety of machines, from personal computers to supercomputers.

2. NFS Version 2 protocol problems

Several problems in NFS Version 2 could only be solved through a new version of the protocol. The 4GB file size limitation has recently become a pressing problem, although implementations of NFS on larger machines such as Cray supercomputers exposed this limitation years ago.

Performance suffers under NFS Version 2 because the protocol requires servers to write data and file system metadata to stable storage (usually disk) synchronously, before replying successfully to a client `WRITE` request *[Ousterhout90]*. The performance problem with synchronous writes was recognized early. NFS Version 2 has an artifact of a proposed interface for asynchronous writes (the undefined `WRITECACHE` procedure).

Implementations have attacked this problem in several ways. *[Moran90]* describes the Prestoserve product, which interposes a software driver between the file system and disk driver to accelerate writes by using nonvolatile RAM. *[Juszczak94]* describes a technique called *write gathering,* which exploits the tendency of more-capable clients to send write requests in clusters to gain parallelism. The author implemented a server that gathers several writes before synchronously committing the data to disk, thereby amortizing the cost of synchronous writes over several requests. *[Hitz94]* describes an integrated file server design that combines a log-based file system and nonvolatile RAM to solve the synchronous write bottleneck.

Some implementations provide an "unsafe" option in their NFS Version 2 server that disables committing modified data to stable storage. While improving performance, this option violates the stable storage guarantee in the NFS Version 2 protocol and can result in data loss. This option has resulted in heated debate.

Lack of consistency guarantees was cited as the cause of excessive requests over-the-wire resulting in increased server loading and response time *[Howard88]*. *[Reid90]* and *[Arnold91]* describe additional problems with NFS Version 2.

3. The NFS Version 3 protocol

Engineers from several companies gathered for a two-week series of meetings in July, 1992, in Boston, MA. to develop an NFS Version 3 specification. The

Presented June 9, 1994
USENIX Summer 1994 — Boston, Massachusetts

group's goal was to address compelling issues in the current protocol that could not be solved by implementation practice. The only absolute requirement was 64-bit file size support.

Other issues under consideration included the following:

- Solving the write throughput bottleneck
- Minimizing the work needed to create an NFS Version 3 implementation given an existing NFS Version 2 implementation
- Ensuring that implementation of the new protocol is feasible on less-capable client operating systems (for example, DOS)
- Completely documenting the resulting protocol and annotating it with implementation examples to aid developers
- Deferring new features to subsequent revisions of NFS due to time constraints

Above all, the driving principles were the following:

- Keep it simple
- Get it done in a year
- Avoid anything controversial

Although it wasn't an absolute requirement, we felt that solving the write throughput bottleneck would provide the most compelling feature.

3.1. Changes introduced

NFS Version 3 represents an evolution of the existing NFS Version 2 protocol. Most of the original design features described in *[Joy84a]*, *[Sandberg85]*, and *[RFC1094]* persist. This revision introduces the following major changes:

- Sizes and offsets are widened from 32 bits to 64 bits.
- The `WRITE` and `COMMIT` procedures allow reliable asynchronous writes.
- A new `ACCESS` procedure fixes known problems with super-user permission mapping and allows servers to return file access permission errors to the client at file open time to provide better support for systems with Access Control Lists (ACLs).
- All operations now return attributes to reduce the number of subsequent `GETATTR` procedure calls.
- The 8KB data size limitation on the `READ` and `WRITE` procedures is relaxed.
- A new `READDIRPLUS` procedure returns both file handle and attributes to eliminate `LOOKUP` calls when scanning a directory.
- File handles are of variable length, up to 64 bytes, as needed by some implementations *[Pawlowski89]*. (We kept the file handle size small enough to allow efficient DOS implementations.)
- Exclusive `CREATE` requests are supported.
- File names and path names are now specified as strings of variable length, with the maximum length negotiated between the client and server (with the `PATHCONF` *[POSIX90]* procedure).
- The errors the server can return are enumerated in the specification—no others are allowed.
- The notion of blocks is discarded in favor of bytes.
- The new `NFS3ERR_JUKEBOX` error informs clients that a file is currently off-line and that they should try again later.

Appendix 1 provides a summary of the protocol differences between NFS Version 2 and NFS Version 3. Refer to *[NFS3]* for more details.

At least eight new versions of NFS have been proposed to fix NFS Version 2, none of which has ever been completely implemented. Public reviews of the draft versions of new protocol specifications have occurred continuously since early 1987. Several changes included in NFS Version 3 first appeared in those eight drafts.

3.2. What was avoided

"Let joy and innocence prevail." *Toys, 1993*

In the years since the NFS protocol was first described, implementation practice solved several problems originally thought to require a protocol revision, although minor, undocumented changes were made to the protocol without a formal revision. In practice, NFS Version 2 mostly works, and we tried not to break it. Accepting common implementation practice reduced the number of changes needed to produce NFS Version 3. Minor protocol changes were cleaned up and incorporated into this work.

We decided to maintain the current stateless design of NFS and not include strict cache consistency. When we defined NFS Version 3, research work on consistent versions of NFS was incomplete. Delaying support for 64-bit file sizes to explore adding stateful consistency was unacceptable. In addition, it seemed clear that supporting strict data consistency introduces complexities that would preclude implementation on less-capable clients. Finally, the recovery benefits of a stateless server were clear, while the issues of stateful recovery were not.

The stateless server design of NFS creates a problem with the replaying of nonidempotent requests. An idempotent request such as `LOOKUP` can be successfully executed any number of times. A nonidempotent request such as REMOVE can be successfully executed only once. Primarily a correctness problem, this condition has been solved through the use of a reply cache of recently serviced requests on the server *[Juszczak89]*. Proposed protocol extensions to NFS attempted to fix this but were essentially misguided. The Boston group simply acknowledged the effectiveness of this implementation technique and left the protocol alone.

Many other changes to NFS Version 2 were proposed in the eight protocol revisions, including the following:

- The ZERO procedure to punch holes in a file
- Append mode writes
- Record-oriented I/O support
- File name to include versions
- User and group fields as strings
- Extended attributes (arbitrary key/value pairs)
- Well-defined UID mapping procedures
- Advisory close procedure
- Resource fork support for the Macintosh
- Multiple OS-dependent name spaces
- A get server statistics procedure

Most of the above proposed features were rejected because by 1992 implementers had worked around purported "protocol limitations" that would prevent implementations on non-UNIX platforms. Other proposed features above were rejected because they were specific to a single operating system. The remaining proposed features were discarded because they attempted to solve a problem simplistically that was best solved correctly (for example, append mode writes versus a full consistency protocol).

4. Design and implementation

NFS Version 3 defines a revision to NFS Version 2; it does not provide a new model for distributed file systems. Because of this, NFS Version 3 resembles NFS Version 2 in design assumptions, file system and consistency model, and method of recovering from server crashes. For a general description of the implementation issues of NFS, see *[Sandberg85]*, *[Israel89]*, *[Juszczak89]*, *[Pawlowski89]*, *[Macklem91]*, and *[Juszczak94]*.

4.1. NFS design

NFS achieves architecture and operating system independence through a strict separation of the protocol and its implementation. The protocol is the interface by which clients access files on a server. A client or server implements the protocol by mapping local file system actions into the file system model defined by NFS. The NFS protocol does not dictate how a server implements the interface or how a client should use the interface *[Satyanarayanan89]*. For example, the NFS Version 3 protocol does not define how a client should manage cached data, but it does provide information to improve cache management.

Although implementations have been used to illustrate aspects of the NFS protocol, the specification itself is the final description of how clients should access servers. Semantic details that were not fully described in the NFS Version 2 specification *[RFC1094]* have proven, in practice, not to be a problem and have been worked out through interoperability testing. Most problems are flaws in implementations, instead of the protocol design.

The NFS protocol is stateless; that is, each request contains sufficient information to be completely processed without regard to other requests. The server does not need to maintain state about any previous requests[1] other than file data on stable storage, and a map of file handles (opaque tokens used by clients to access files) to files derived from file system data. Of course, most servers cache file data that has been synchronized to disk to improve performance. However, this cached data is not needed for correct operation.

Server crash recovery is simple. A client need only retry a request until the server responds; the client does not know that the server has rebooted (although the user may notice delayed responses). Experience at Sun with *network disk (nd)*, an earlier method of sharing disk storage on a network, led to the stateless server requirement in the initial design of NFS *[Joy84b]*.

The NFS Version 3 protocol requires that modified data on the server be flushed to stable storage before replying. Only asynchronous writes are excepted. NFS clients block on *close(2)* until all data is flushed to stable storage on the server, to return any errors to the application that might occur during delayed writes (for example, out of space).

NFS clients are decidedly not stateless. NFS clients hold modified data that has not been flushed to the server as well as cache file handles and attributes. Clients typically use attribute information, such as file modification time, to validate cached information. When a client crashes no recovery is necessary for either the client or the server.

[1] To be precise, the reply cache on a server contains volatile state needed for correctness *[Kazar94]*. See *[Bhide91]* for further discussion on the reply cache and its implications for server correctness. TCP-based implementations of NFS still need a reply cache to prevent destructive replay following connection re-establishment.

Thus, NFS servers are stupid and NFS clients are smart. NFS Version 3 offers the possibility of potentially smarter clients.

4.2. Multiple version support

The Remote Procedure Call (RPC) protocol provides explicit support for multiple versions of a service *[RFC1057]*. The client and server implementations of NFS Version 3 provide backward compatibility with NFS Version 2 by supporting *both* NFS Version 2 and NFS Version 3. By default, an RPC client and server bind using the highest version number they both support. Client or server implementations that cannot support both versions (for example, due to memory restrictions) should support NFS Version 2.

4.3. Implementation issues

A primary goal in restricting the changes between NFS Version 2 and NFS Version 3 was to minimize new implementation issues. Implementation issues exist in the following areas:

- 64-bit file sizes and offsets
- Asynchronous writes
- `READDIRPLUS`—read directory with attributes
- `NFS3ERR_JUKEBOX`
- Weak cache consistency
- Other issues

4.3.1 64-bit file sizes and offsets

The 64-bit extensions in NFS Version 3 introduce problems with mismatched clients and servers, such as a 32-bit client and a 64-bit server, or a 64-bit client and a 32-bit server.

A 64-bit client will never encounter a file that it cannot handle when using a 32-bit server. If it sends a request that the server cannot handle, the server should return `NFS3ERR_FBIG`.

The problems posed by a 32-bit client and a 64-bit server are more difficult. The server can handle anything that the client can generate. However, the client cannot handle a file whose size can not be expressed in 32-bits, and will not properly decode the size of the file into its local attributes structure. One solution is for the client to deny access to any file whose size cannot be expressed in 32 bits. This introduces anomalous behavior when a file is extended by the client beyond its limit, thus rendering the file inaccessible.

Another solution is for the client to map any size greater than it can handle to the maximum size that it can handle, effectively "lying" to the application program. This allows the application access to as much of the file as possible given the 32-bit offset restriction. Although this solution eliminates the anomalous behavior described in the first solution, it introduces the problem that a client might be able to access only part of a file. However, other solutions exist.

4.3.2 Asynchronous writes

NFS Version 3 asynchronous writes eliminate the synchronous write bottleneck in NFS Version 2. When a server receives an asynchronous `WRITE` request, it is permitted to reply to the client immediately. Later, the client sends a `COMMIT` request to verify that the data has reached stable storage; the server must not reply to the `COMMIT` until it safely stores the data.

Asynchronous writes as defined in NFS Version 3 are most effective for large files. A client can send many `WRITE` requests, and then send a single `COMMIT` to flush the entire file to disk when it closes the file. This allows the server to do a single large write, which most file systems handle much more efficiently than a series of small writes. For very large files, the server can flush data in the background so that most of it will already be on disk when the `COMMIT` request arrives.

Asynchronous writes are optional in NFS Version 3, and specific client or server implementations can choose not to support this feature. A server can choose to flush asynchronous write requests to stable storage. In this case, the server indicates this in the `WRITE` reply. Clients with insufficient memory to support the necessary buffering required for server crash recovery can always request synchronous writes.

4.3.2.1 Crash recovery

The design of asynchronous writes is consistent with the stupid server and smart client philosophy of NFS. The client is required to keep a copy of all uncommitted data to support recovery following a server crash. The replies for `WRITE` and `COMMIT` requests include a *write verifier* that clients use to detect server crashes. The write verifier is an 8-byte value that the server must change whenever it crashes. Servers commonly use their boot time as a write verifier, because it is guaranteed to be unique after each crash. The client must save the write verifier returned by each asynchronous `WRITE` request and compare it to the write verifier returned by a later `COMMIT` request. If the write verifiers do not match, then the client assumes that the server has crashed and rebooted.

The client must then rewrite all uncommitted data. Clients can push data with synchronous writes following server failure. The client can delay rewriting data when it detects a crash to avoid flooding a newly rebooted server with `WRITE` requests. Figure 1 shows

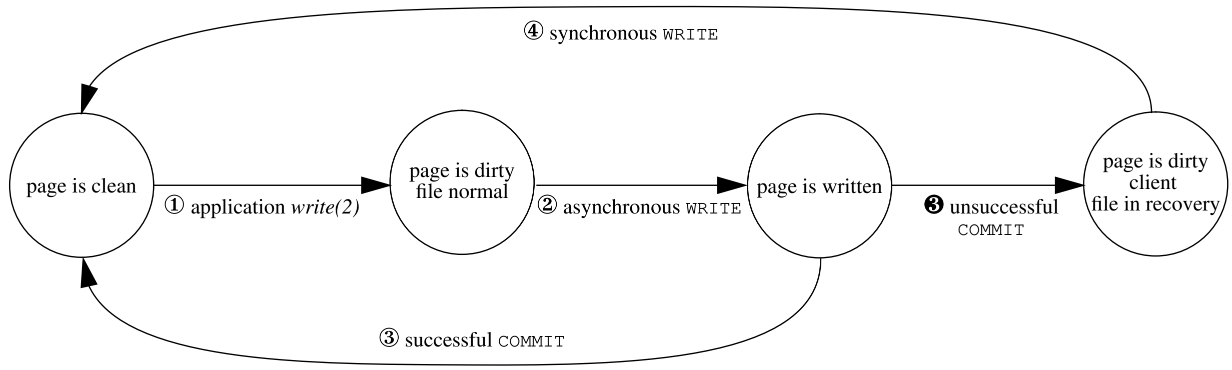


Figure 1. Client page states with asynchronous writes (The Digital OSF/1 implementation). This diagram shows the state changes that occur as a page of memory containing file data is modified, written, and then committed. ① A local application modifies the page, and it is marked dirty. ② The client asynchronously writes the data to the server. The client stores the write verifier from the asynchronous write request with each page. An explicit *msync(2)*, *fsync(2)* or *close(2)* from the application, a file system sync, or page reclamation will trigger a `COMMIT`. The write verifier returned from the `COMMIT` request is compared against those stored with the written pages. ③ The page's write verifier matches the returned verifier, and the commit succeeds. ❸ The write verifier for the page does not match the returned write verifier, triggering recovery. ④ The client synchronously writes the data to the server.

the state changes that occur as a page of memory containing file data is modified, written to the server, and then committed.

4.3.2.2 Server details

An NFS Version 3 server makes the following three guarantees:

- For a synchronous `WRITE` request, the server will commit to stable storage all data and modified metadata.
- The server will not discard uncommitted data without changing the write verifier.
- The server will commit the file's data and modified metadata to stable storage for the range specified in the `COMMIT` request before reporting success.

Other conditions arise in which the write verifier must change. For example, the server must change the write verifier on failover if NFS Version 3 forms the basis of a non-shared memory, highly available implementation of NFS *[Bhide91]*. The unsynchronized data is not available to the backup processor, and there is no guarantee that the primary processor was able to flush uncommitted data to stable storage before going down.

If a server is shut down cleanly, it could be advantageous to save the write verifier for reuse when the server is brought back on line. This avoids triggering client rewrites of already committed data.

4.3.2.3 Data sharing

Asynchronous writes make write sharing without using a higher-level application synchronization protocol even less attractive than with NFS Version 2. NFS Version 3 clients preserve close-to-open consistency: clients typically block on a *close(2)* until all data is flushed to server stable storage and revalidate cached data with an attribute check on *open(2)*. Strictly speaking, close-to-open consistency is only an implementation practice. Data sharing semantics of NFS Version 3 differ from those of NFS Version 2 if an NFS Version 3 server reboots and loses uncommitted data. Because write sharing between NFS Version 2 clients was never supported in the absence of locking, changes in essentially undefined behavior is not considered a major issue.

4.3.3 **`READDIRPLUS`**

NFS Version 3 contains a new operation called `READDIRPLUS`, which returns file handles and attributes in addition to the directory information returned by `READDIR`.

`READDIRPLUS` exploits observed request sequences generated by NFS Version 2 clients. For example, when a UNIX user types "**`ls -F dir`**" to browse a directory containing 20 entries, the **`ls`** command opens the target directory, reads it, and then calls *stat(2)* 20 times. In NFS Version 2, a `READDIR` request would be followed by 20 sequential `LOOKUP` requests to retrieve attributes (and file handles). In NFS Version 3, a single `READDIRPLUS` retrieves the name list and attributes for the 20 entries, significantly reducing the command execution time.

There are some drawbacks to READDIRPLUS, however. A READDIRPLUS is more expensive than a corresponding READDIR. Results from an implementation that generates exclusively READDIRPLUS requests show a performance drop because attributes for all entries in a directory are fetched repeatedly for every access to a directory.

The READDIRPLUS operation can be viewed as a way to get the contents of a directory and to populate name and attribute caches for the entries in that directory at the same time. The READDIRPLUS operation should be used only when reading a directory for the first time or when rereading a directory whose cache entry was invalidated because the directory was modified. A READDIRPLUS should not be issued when a valid cache entry for a directory exists, because it is likely that a READDIRPLUS operation was recently issued to populate the various caches with directory entry attributes and file handles.

4.3.4 NFS3ERR_JUKEBOX

NFS3ERR_JUKEBOX[2] lets servers inform clients that a file is temporarily inaccessible (archived offline or locked against modification for backup) and that they should retry the request later. It is intended to improve the behavior of NFS in hierarchical storage management applications.

In NFS Version 2, a server performs one of three actions if a file is temporarily inaccessible. The first action is to drop the request, which forces the client into normal back-off and retransmission. The request will be satisfied at some later time on a retry. The second action is to have the server block a service thread until the file again becomes accessible. The second action is often implemented inadvertently; because clients employ mechanisms like biods to gain parallelism and will emit several related requests to one file, blocking server threads can hang the server. The third action is to return some error to the client, thus rejecting the request.

An NFS Version 3 server returns NFS3ERR_JUKEBOX when a file is temporarily inaccessible. The client operating system does not return the error to the application but handles it internally by aggressively delaying reissue of the request, thereby reducing server load due to request retransmission. After a tunable delay, the request is reissued. The client should reissue the request with another transmission id.

[2] The term "JUKEBOX" is a long standing joke in the NFS community. We kept the historical error name even though it incorrectly implies a binding to a particular HSM mechanism. Given the generic intent of the error, NFS3ERR_TMP_INACCESSIBLE would be more appropriate.

4.3.5 Weak cache consistency

Many NFS Version 2 clients cache file and directory data to improve performance. To determine whether cached data is valid, a client sends a GETATTR request. If the new modification time from the server matches the modification time in the client's cached attributes, then the client assumes its cache is up-to-date. If the modification times don't match, then the file must have changed, and the client invalidates its cache.

This method fails when the client itself modifies the file being cached. For example, if a client writes to one part of a file, cached data for other parts is probably still valid. But it is impossible for the client to be sure, because the client's own WRITE request updated the file's modification time. A reckless client might keep the cache data (which is dangerous), and a cautious client might invalidate the cache (which is slow).

Weak cache consistency offers an alternative by helping clients determine more accurately when to invalidate their cache. The reply for each NFS Version 3 request that can modify data includes two versions of the file's attributes: pre-operation attributes from just before the server performed the operation and post-operation attributes from just after the operation. If the modification time in the pre-operation attributes from the server matches the cached attributes on the client, then the client's cache is valid. The client should update its attribute cache with the new post-operation attributes.

Weak cache consistency does not provide true consistency such as found in Sprite *[Nelson88]*. With weak cache consistency, clients might see an inconsistent view of server data. For example, one client might have modified a file locally but not yet flushed the new data to the server. Even if it has, a second client will only verify modification times when a file is first opened or when the cached attributes time out. As a result, a second client's cache may be out of date.

Some servers may be unable to generate pre-operation attributes, so clients should be prepared to fall back to NFS Version 2 behavior. Since weak cache consistency is just a hint, client implementations are free to use it or ignore it.

4.3.6 Other issues

Two changes in NFS Version 3 impose extra work on the client. For many NFS Version 3 requests, it is optional to return file handle and attribute information that is mandatory in NFS Version 2. For example, in

NFS Version 2, the `CREATE` request must return the file handle and attributes for the newly created file, but in NFS Version 3, their return is optional. As a result, an NFS Version 3 client must be prepared to issue a `LOOKUP` after each `CREATE`, in case the server does not return a file handle for the new file. Furthermore, in NFS Version 3, it is optional for `LOOKUP` to return attributes, so the client must also be prepared to issue a `GETATTR`.

NFS Version 2 servers are required to accept all or none of the data in a `WRITE` request. In NFS Version 3, a server can accept only some of the data in a write, and the client is expected to send the rest a second time. For example, a client might send an 8192 byte request, but a server might choose to accept only 1 byte. The client must be prepared to send the remaining 8191 bytes a second time, and again, the server might choose not to accept the entire request.

In practice, these features are unlikely to be a problem because most server implementations will always return optional information and accept the entire contents of `WRITE` requests.

4.4. Changes to related protocols

NFS Version 3 continues the philosophy of building a network file service from a collection of cooperating protocols. The mount protocol (MOUNT) allows an NFS client to gain access to an exported directory on a server, and the network lock manager protocol (NLM) supports remote file locking for NFS.

Changes to the file handle and file size fields in NFS Version 3 required corresponding changes in MOUNT and NLM, so new versions of both protocols have been released. The new MOUNT specification allows a successful mount to return a list of acceptable RPC authentication flavors (such as DES or Kerberos) for the client to use. Automounter facilities can use this information to correctly access servers which require certain flavors of authentication. The new MOUNT protocol is also slightly cleaner than the previous one. For example, legal error values have been enumerated instead of allowing any UNIX error number.

5. Performance

A major goal of NFS Version 3 was to improve performance, especially in write throughput. Performance was improved by the following:

- Providing reliable asynchronous writes
- Removing the 8KB data size limitation for `READ` and `WRITE` requests
- Providing a `READDIRPLUS` procedure that returns file handles and attributes with directory names
- Returning attribute information in all replies
- Providing weak cache consistency data to allow a client to more effectively manage its caches

5.1. Test setup

We measured Digital's OSF/1 implementation of NFS Versions 2 and 3. The local file system employed for these tests was the Berkeley Fast File System with enhanced clustering *[McVoy91]*. Except where noted, the following configuration was used to generate the performance results:

- Two Digital Model 3000/600 96MB workstations
- Private FDDI network
- Server running 16 nfsds (multiple threads of execution used on an NFS server to gain parallelism).
- Client running 7 nfsiods (or biods—multiple threads of execution used on an NFS client to gain parallelism)
- With and without Prestoserve on server, using 1MB NVRAM
- With and without write gathering on server
- Server configured with one 1GB RZ26 SCSI disk, 2.3 MB/sec raw transfer rate.

The tests ran with NFS running on top of UDP with a maximum transfer size of 8KB. The larger transfer sizes permitted by NFS Version 3 were not exploited. Measurements at SunSoft on a system using larger than 8KB transfer sizes showed improved write throughput, presumably from the reduced file system overhead resulting from fewer separate I/O requests and fewer RPC messages over-the-wire.

5.2. Sequential write throughput

Figure 2 shows the results of writing a 10MB file over a private FDDI network using NFS Version 2 and NFS Version 3 protocols and varying the server configuration to enable/disable Prestoserve acceleration and server write gathering. We consider the NFS Version 2, no write gathering, no Prestoserve configuration to be the average NFS write throughput available today. We believe that the NFS Version 2, write gathering, Prestoserve configuration provides competitive NFS write throughput. We observe the following:

- NFS Version 2 with Prestoserve and NFS Version 3 delivers the maximum raw device rate to the remote client.
- NFS Version 3 with asynchronous writes at 2323 KB/s delivers only 1% less throughput than NFS Version 2 with Prestoserve and write gathering at 2346 KB/s, but it consumes 36% less server CPU.

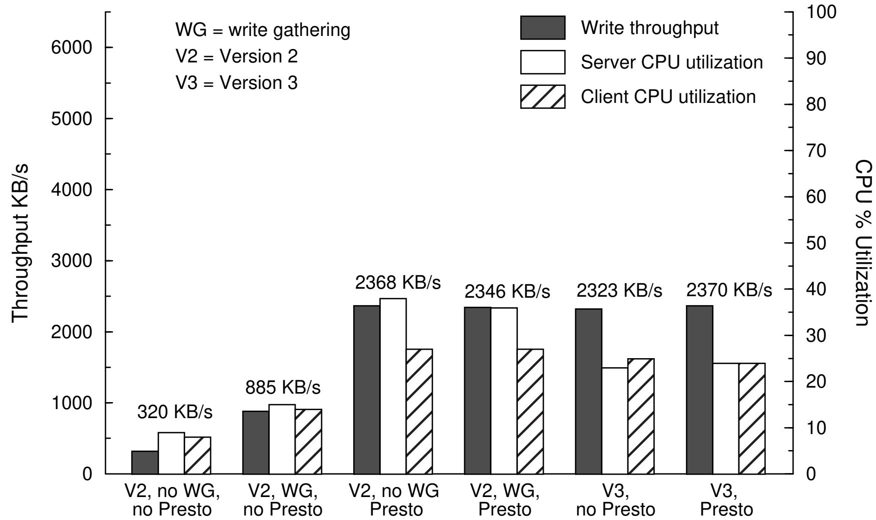


Figure 2. Comparisons of 10MB file writes over FDDI, Digital OSF/1, Digital 3000/600

- At 2323 KB/s, NFS Version 3 is seven times faster than NFS Version 2 at 320 KB/s for a typical configuration with no write gathering and no Prestoserve.

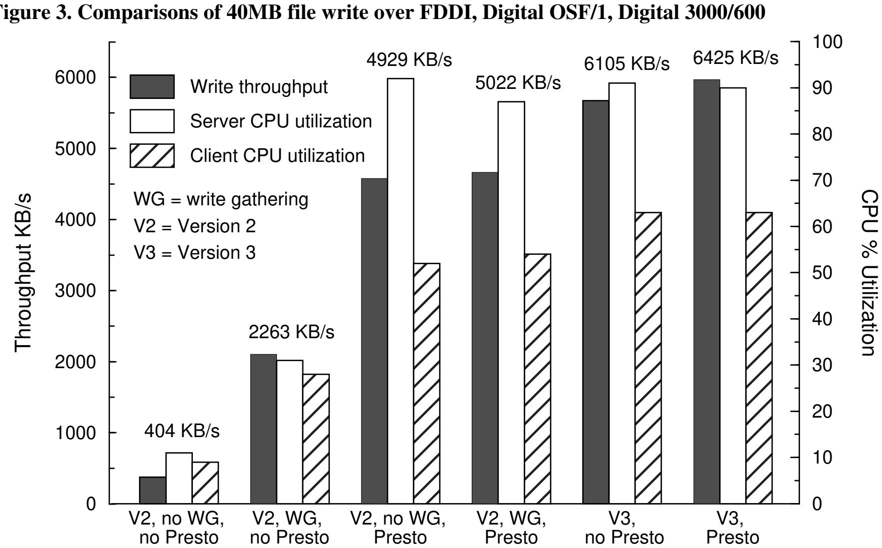
The NFS Version 3 client emitted only asynchronous writes in these tests; therefore, server write gathering had no effect. This configuration is not shown. Prestoserve further improves NFS Version 3 asynchronous writes because there is a synchronous component to writing metadata during local file system clustering. NFS Version 2 with Prestoserve provides higher throughput on a single disk system than NFS Version 3, because Prestoserve masks the cost of reduced cluster transfer sizes and missed rotations seen in its absence. Multiple spindles can help mask these effects in the absence of accelerator hardware.

It was clear that the disk was the bottleneck for the above test, given the low CPU utilizations, available network bandwidth on FDDI (100 Mbit/s), and the raw speed of the disk. To remove the disk bottleneck, we made a second set of runs, sequentially writing a 40MB file, with the following configuration changes:

- Server configured with four 2GB RZ28 SCSI disks, each 4.8 MB/sec raw transfer rate, four-way striped
- Client running 15 nfsiods (or biods)

The results in Figure 3 show that for sufficiently large files on a non-disk bound server, NFS Version 3 delivered 6105 KB/s, compared to an NFS Version 2 server with Prestoserve and write gathering that delivered 5022 KB/s. NFS Version 3 delivered 22% more throughput at a similar server CPU utilization. The

Figure 3. Comparisons of 40MB file write over FDDI, Digital OSF/1, Digital 3000/600



Test	Basic test description	Version 2 NFS no Presto	Version 3NFS no Presto	Version 2 NFS with Presto	Version 3 NFS with Presto
1	File and directory creation create 155 files 62 directories 5 levels deep	8.39	8.21	0.87	0.77
2	File and directory removal remove 155 files 62 directories 5 levels deep	3.71	3.66	1.60	1.20
3	lookups across mount point 500 getwd and stat calls	0.81	0.81	0.74	0.71
4	setattr, getattr, and lookup 1000 chmods and stats on 10 files	11.18	11.18	0.94	1.00
5a	write 1MB file 10 times throughput	12.00 869 KB/s	5.35 1957 KB/s	4.68 2238 KB/s	4.69 2234 KB/s
5b	read 1MB file 10 times throughput	1.48 7056 KB/s	1.48 7052 KB/s	1.49 7019 KB/s	1.47 7128 KB/s
6	readdir 20500 entries read, 200 files	2.87	2.79	1.40	0.96
7	link and rename 200 renames and links on 10 files	6.71	6.71	1.22	1.10
8	symlink and readlink 400 symlinks and readlinks on 10 files	6.73	6.70	1.25	0.98
9	statfs 1500 statfs calls	0.92	1.50	0.92	1.40
Basic tests NFS RPC count		13166	11032	13166	11032
Total NFS RPC count for Basic, General and Special tests		21865	17764	21865	17764

Table 1: Connectathon Basic test suite results, 7 biods, single disk spindle, (results in seconds, except as noted)

maximum throughput of 6425 KB/s was achieved with NFS Version 3 and Prestoserve. Throughput increased with this configuration change, but not to the point of the disk bandwidth limitation or CPU exhaustion. The bottleneck moved to the network because of the limited number of stations, limited application parallelism, and FDDI token holding time characteristics of the network interfaces.

We conclude that asynchronous writes improve both client throughput and server efficiency. They provide most of the benefits associated with running an NFS Version 2 server in "unsafe" mode, while ensuring data reliability after server failure[3]. Prestoserve should still accelerate small file writes, as well as other modifying requests like `CREATE` and `REMOVE`.

5.3. Connectathon test suite results

Because the LADDIS benchmark generates NFS Version 2 RPC calls directly to measure server performance *[Wittle93]*, it cannot measure NFS Version 3 without modification. As an alternative, we ran the Connectathon test suite, which was developed to test the interoperability of NFS implementations. It runs on the client on a remotely mounted directory and exercises both client and server NFS code. It consists of three passes that cover the basic functionality of a file system. The Basic pass isolates specific features of the client file system, and consists of ten separate tests. Testing a single client file system feature typically generates a mix of NFS requests. The General pass runs multiple simultaneous large compiles, as well as *nroff(1)*. The Special pass exercises boundary cases in NFS operations.

Table 1 contains the results of running the Connectathon test suite. We conclude the following from these results:

- Again, NFS Version 3 asynchronous writes are clearly a win (see test 5a).
- Prestoserve remains useful on the server for other metadata operations (`CREATE`, `REMOVE`, etc.), as shown by tests 1, 2, 4, 6, 7 and 8. Test 6 performs file deletions in addition to reading directory entries, which explains the improvement with Prestoserve.
- NFS Version 3 reduces the total number of RPC messages by 18% compared to NFS Version 2. The reduction is due entirely to the increased frequency of returned attributes and better cache management through weak cache consistency data. This reduction more than offsets the calls to the new `ACCESS` and `COMMIT` RPC procedures.

[3] *[Nelson88b]* suggests that unsafe writes would provide greater throughput than asynchronous writes with close-to-open consistency. That is, assuming that `COMMIT` blocks until all remaining data is on disk when a file is closed, unsafe mode implementations which do not block would clearly perform better. For large files, this effect should be minimal.

NFS Version 2

```
calls
21865
null      getattr   setattr   root      lookup    readlink  read
0   0%    4058 18%  1168  5%  0   0%    6954 31%  250   1%  1779  8%
wrcache   write     create    remove    rename    link      symlink
0   0%    1881  8%  675   3%  1175  5%  352   1%  250   1%  250   1%
mkdir     rmdir     readdir   statfs
173   0%  173   0%  972   4%  1755  8%
```

NFS Version 3

```
calls
17764
null      getattr   setattr   lookup    access    readlink  read
0   0%    1282  7%  1168  6%  5499 30%  309   1%  250   1%  1731  9%
write     create    mkdir     symlink   mknod     remove    rmdir
1881 10%  675   3%  173   0%  250   1%  0   0%    1175  6%  173   0%
rename    link      readdir   readdir+  fsstat    fsinfo    pathconf  commit
352   1%  250   1%  758   4%  18    0%  1755  9%  0   0%    0   0%    65   0%
```

Figure 4. Detail of RPC counts for all three passes of the Connectathon Test Suite

The read throughput results from test 5b reflect over-the-wire data transfers. Test 5b was modified to use the *mmap(2)* system call to invalidate the client's data cache, forcing the requests to go over-the-wire. However, the data was cached on the server. The detailed RPC counts for the NFS Version 2 and Version 3 results are shown in Figure 4.

5.4. *find(1)* results

The *find(1)* command was used to measure the effect of `READDIRPLUS`. *find(1)* scanned a remote file tree containing 9612 files distributed over 155 directories that were up to seven levels deep. The results are shown in Table 2. The over-the-wire byte counts include all protocol headers.

Using `READDIRPLUS` to fetch file handles and attributes of entries in a directory reduces the *find(1)* execution time by 36%, compared to NFS Version 2. Reduced execution time can be attributed primarily to the tenfold reduction in over-the-wire messages. The 155 `GETATTR` requests are generated to ensure close-to-open consistency when opening a directory. Using the `READDIRPLUS` procedure in NFS Version 3 reduced the total bytes transferred over-the-wire by 43% and the cumulative server CPU (percent utilization $\times$ elapsed time) by 46%, compared to using the `READDIR` and `LOOKUP` procedures in NFS Version 2. `READDIRPLUS` is clearly a win in this example.

The test was rerun with `READDIRPLUS` disabled in NFS Version 3. The last column in Table 2 shows these results. Disabling `READDIRPLUS` increases execution time by 95%, compared to the NFS Version 3 result with `READDIRPLUS` enabled. More disturbing, execution time increased by 24%, compared to the NFS Version 2 results. We attribute this to the new `ACCESS` procedure and to larger message sizes in NFS Version 3, which increased the total bytes transferred

Table 2: *find(1)* results

	NFS Version 2		NFS Version 3 w/ `READDIRPLUS`		NFS Version 3 w/o `READDIRPLUS`	
real time	9.9s		6.3s		12.3s	
client system time	4.8s		3.1s		4.6s	
`GETATTR` count	155	1%	155	13%	155	1%
`LOOKUP` count	10076	95%	155	13%	10076	92%
`ACCESS` count	n/a		310	27%	310	2%
`READDIR` count	173	1%	0	0%	181	1%
`READDIRPLUS` count	n/a		358	31%	0	0%
`STATFS`/`FSSTAT` count	155	1%	155	13%	155	1%
Total count	10559	100%	1133	100%	10877	100%
bytes sent	2108523		225209		2214575	
bytes received	1973952		2088284		3270824	
total bytes over the wire	4082475		2313493		5485399	
client CPU utilization	53%		42%		44%	
server CPU utilization	37%		32%		31%	

by 34% when compared to NFS Version 2. Message sizes increased because new fields were added and old fields were widened.

This result illustrates a fundamental tradeoff in the NFS Version 3 design: increased RPC request and reply sizes are to be offset by new features in the protocol. Naive implementations that fail to use the new features will perform worse for some benchmarks than NFS Version 2, but effective use of new features will increase overall performance.

6. Cost of porting

The Digital OSF/1 implementation illustrates the effort and cost to port the SunSoft NFS Version 3 reference source into an existing Version 2 implementation. The source code size of an implementation that supports both protocols is roughly 30,000 lines (C code + comments + white space). The Version 2 and Version 3 specific portions of the total are about 12,000 lines each, with 6,000 lines of shared subroutines. Assuming engineers familiar with NFS Version 2, the effort needed to produce an implementation that supports both versions of the NFS protocol for initial testing is the following:

server	1 person-month
client (excluding asynchronous writes)	2 person-months
client asynchronous writes	1 person-month

Digital's OSF/1 based kernel uses a unified page cache managed by the virtual memory subsystem for both program text and file data. This complicated the client implementation of asynchronous writes because of dependencies on data structures and interfaces in the virtual memory system.

7. Related work

"Look on my works, ye Mighty, and despair!"
Ozymandias, *Shelley, 1817*

The NFS Version 3 protocol mitigates the need for NFS-specific write gathering techniques on clients that support asynchronous writes, because a server can now simply process clusters of related asynchronous writes as part of its local buffered file system activity *[McVoy91]*. However, NFS-specific write-gathering on servers is still useful in supporting less-capable NFS Version 3 clients that do not support asynchronous writes or more-capable clients that resort to synchronous behavior during recovery. The stable storage semantics for metadata modifying operations, such as `CREATE`, remain unaffected by NFS Version 3. Thus, a server can still benefit from fast stable storage. To a lesser extent, fast stable storage techniques still improve asynchronous `WRITE` performance, especially for small files.

Adaptive retransmission strategies to improve the behavior of NFS over UDP (as described in *[Nowicki89]*, derived from *[Jacobson88]*) and the use of TCP to improve performance over wide area networks *[Macklem91]*, are applicable to NFS Version 3. NFS Version 3 relaxes the 8KB limitation on the data portion of a `READ` or `WRITE` request, permitting more efficient use of TCP.

Three efforts to revise the NFS protocol are related to this work. The first is Spritely NFS, described in *[Srinivasan89]*, *[Mogul92]*, and *[Mogul93]*. Spritely NFS uses a stateful server that controls client caching behavior to ensure consistency. State recovery following a crash is server-driven. The server keeps a nonvolatile list of old clients that are contacted during a grace period following reboot to initiate the rebuilding of state on the server. Spritely NFS employs consistency to address performance issues in NFS Version 2 by allowing clients to defer writes and by eliminating the need for clients to poll the server to detect file changes.

The second effort is NQNFS *[Macklem94]*, which defines extensions to NFS Version 2 that are similar to those found in NFS Version 3. Size and offset fields were widened to 64 bits, and a `READDIRPLUS` procedure was added. Time-based leases provide a mechanism for data consistency and cache coherence among clients. Clients need to anticipate lease expiration. Clients do not have special recovery code. Instead, leases are short enough to expire while the server is rebooting, forcing clients to request renewals (thereby driving recovery) from the newly rebooted server. On reboot, a server accepts only writes during a grace period, after which it will grant new leases.

While the results of both NQNFS and Spritely NFS looked promising at the time we defined NFS Version 3, both were unfinished. We decided that adding consistency to NFS was contrary to our minimalist goals and best left for a subsequent revision.

The third effort, *[Fadden92]* and *[Glover92]*, described Trusted NFS (TNFS), which defines a method for handling ACLs and data labels that conserves space. Acknowledging that security data can be large, TNFS maps the data into opaque tokens and requires a separate token mapping service to convert to and from a canonical over-the-wire format. We decided not to incorporate this work into NFS Version 3 because of instability in the POSIX ACL specification and the relative immaturity of extant implementations of TNFS.

DCE DFS *[Kazar90]* is related to NFS Version 3 only in that it describes an amount of effort that we clearly did not want to undertake. Our primary goals were to improve NFS Version 2 and deploy a new version quickly. We preferred to retain the ease of server crash recovery, at the expense of not supporting some of the more valuable features of DCE DFS.

8. Future work

The strategy for using `READDIRPLUS` needs further research. Reading the contents of a very large directory with `READDIRPLUS` can eject potentially more valuable entries from client caches. Finding heuristics to guide choosing between `READDIR` and `READDIRPLUS` is hard because an NFS client cannot tell whether an application will need attribute information for a directory's children or not. More experience could lead to better heuristics than the simple ones used now.

An NFS Version 3 client trying to do effective cache management with weak cache consistency requires that the server guarantee atomicity of modifying operations and pre- and post-operation attribute generation. The performance cost of supporting such atomicity on the server is not fully understood, particularly for multiprocessor server implementations where extensive locking could result in unwanted serialization. More analysis is needed. Weak cache consistency with the `WRITE` procedure provides no useful sharing semantic.

Additional characterization and tuning of NFS Version 3 under more complex workloads is needed. An NFS Version 3 LADDIS benchmark is needed. Tuning NFS Version 3 implementations should not pose insurmountable problems.

We did not expect the NFS Version 3 specification to be perfect. Our hope is that the protocol specification will grow to reflect common practice and provide guidelines on conforming behavior. The development of an NFS Version 3 Validation Suite by SunSoft will aid interoperability. Finally, interoperability testing of implementations at Connectathon remains the cornerstone of successful file sharing with NFS.

8.1. NFS Version 4

In defining NFS Version 3, we assumed that other protocol revisions would follow, allowing us to defer features. Improved data and cache consistency is an obvious candidate for NFS Version 4. POSIX write-sharing semantics exist today on a single NFS client. NFS Versions 2 and 3 support a client-driven bounded time-based model for write sharing *[Kazar88]*, with close-to-open consistency. This model does not provide sufficient guarantees for concurrent write-sharing between cooperating clients in the absence of explicit locking. The fact that write-sharing is infrequent even in those distributed file systems that support it *[Welch90]* is a reason NFS has been successful despite this limitation. Both Spritely NFS and NQNFS demonstrate how to provide stronger consistency guarantees with a provision for server and client crash recovery. Both approaches depend on the clients to re-establish state after server reboots.

Disconnected operation of fixed and nomadic clients is a potential area for future work. More investigation is required on how consistency guarantees work, if at all, in the presence of clients disconnected longer than the lease terms or callback timeouts used by NQNFS or Spritely NFS, respectively.

Stronger security models in NFS are another area for future work. More research is needed on whether to pursue trusted system support in general.

The problems of consistent name space construction and increased availability are areas of research for future protocol revisions and are perhaps best solved with innovative implementations using existing protocols.

9. Conclusions

The constrained NFS Version 3 effort addressed the following concerns with NFS Version 2:

- 64-bit file sizes are now supported.
- Asynchronous writes increased throughput sevenfold over unaccelerated NFS Version 2 implementations.
- Over-the-wire traffic measured both by RPC counts and network loading has been reduced.
- Directory browsing is faster, with less network loading and lower CPU utilization.
- Performance improvements were achieved despite the size increase of the file attribute structures resulting from 64-bit file size support.
- Many "minor annoyances" of the NFS Version 2 protocol have been corrected.

NFS Version 3 was specified, reviewed, prototyped, verified, and supplied by multiple vendors for external testing in less than 24 months from the initial Boston meetings. At Connectathon in 1993, prototype implementations interoperated successfully. We achieved the goal of providing measurable improvements over NFS Version 2 with little effort required to create an implementation.

There is more work to be done. NFS Version 3 offers the potential for better name and attribute cache management than is possible with NFS Version 2. Realization of this potential is a current and future effort.

9.1. Availability

The NFS Version 3 protocol specification draft can be obtained from bcm.tmc.edu, gatekeeper.dec.com and ftp.uu.net using anonymous FTP.

NFS Version 3 will be available in the next major release of Digital's OSF/1. Servers will fully support NFS Version 3, as well as provide NFS Version 2 for interoperability with older clients. At SunSoft, a Solaris 2 implementation of NFS Version 3 that supports TCP and large transfer sizes is in early deployment and will shortly go to external field test. In addition, a reference implementation of NFS Version 3 with TCP support is undergoing final testing. Early access to the reference implementation from SunSoft will occur this summer. Other implementations are in progress. Contact your vendor for further information.

SunSoft is developing an NFS Version 3 Protocol Validation Suite to provide a tool to help ensure interoperability of clients and servers. This validation suite will be made available for licensing.

10. Acknowledgments

Rusty Sandberg was the author of the earliest NFS Version 3 proposal. Eight specifications intervened between then and now; in many ways we returned to the simplicity of the original. Peter Staubach designed the versions of asynchronous writes, NFS3ERR_JUKEBOX, and READDIRPLUS described in this paper. Peter also introduced the notion of weak cache consistency in NFS. The Boston group included Cathe Ray, Carl Smith, Peter Staubach, and Brian Pawlowski of SunSoft, Inc., Fred Glover, and Chet Juszczak of Digital, Mark Wittle of Data General, John Gillono of Cray Research, Tom Talpey of OSF, and Geoff Arnold of SunSelect, Inc. Spencer Shepler of IBM would have joined us but for his wedding; he did participate in the post-Boston discussion. Chris Duke has been very supportive in his role as the Sun NFS engineering manager. Charlie Briggs at Digital reviewed early drafts and suggested the state diagram. Eric Werme at Digital worked on the server implementation. Michael Kupfer implemented the Network Lock Manager reference source. Brent Callaghan implemented the original MOUNT reference source. Glen Dudek at VGI, our paper shepherd for USENIX, provided invaluable detailed reviews beyond the call of duty. Jeffrey Mogul commented on a very rough early draft. Chris Duke, Byron Rakitzis, Rob Salmon, Dana Treadwell, Rusty Sandberg, Kim Pawlowski, Olga Koudalides, Ellie Koudalides, Michael Eisler, Brian Ehrmantraut, Michael Nelson, Tom Talpey, Bob Lyon, Cheena Srinivasan, Eric Werme, Michael Kupfer and Tom Tierney reviewed various drafts of this paper on an outrageously compressed schedule. Chad Davies, Richard Binder, and Karla Sorenson greatly improved the readability of this paper. Mike Kazar kept the paper honest.

11. Bibliography

[Arnold91] Arnold, Geoff., "Change and Non-change in NFS," *Proc. of European Sun Users Group*, 1991. *Discusses changes requiring a protocol revision in NFS.*

[Bhide91] Bhide, A., Elnozahy, E., Morgan, S., "A Highly Available Network File Server," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes an NFS server implementation using redundant servers and dual-ported disks that logs volatile reply cache information to disk.*

[Fadden92] Fadden, Fran, "Token Mapping Service," Trusted System Interest Group, TSIG document TSIG-TNFS-006.01.01, May 24, 1992. *Description of the security token mapping scheme proposed in TNFS.*

[Glover92] Glover, Fred, "Request for Comments on a Specification of Trusted NFS (TNFS) Protocol Extensions," Trusted System Interest Group, TSIG document TSIG-TNFS-001.02.02, May 24, 1992. *Proposed draft standard for security extensions to NFS for a trusted environment.*

[Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a highly integrated approach using a log-based file system and nonvolatile RAM to solve the write bottleneck on NFS Version 2.*

[Howard88] Howard, J.H., M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems* 6(1). February, 1988. *Primary reference on the Andrew File System—contrasts the performance and scalability of AFS and NFS—cites the lack of consistency guarantees as the cause of poor scalability of NFS file servers.*

[Israel89] Israel, Robert K., Sandra Jett, James Pownell, George M. Ericson, "Eliminating Data Copies in UNIX-based NFS Servers," *Uniforum Conference Proceedings*, San Francisco, CA, Feb. 27 - Mar. 2, 1989. *Describes two methods for reducing data copies in NFS server code.*

[Jacobson88] Jacobson, V., "Congestion Control and Avoidance," *Proc. ACM SIGCOMM '88*, Stanford, CA, August 1988. *Describes TCP performance improvements over WANs and gateways. This work was a starting point for the NFS Dynamic Retransmission work.*

[Joy84a] Joy, Bill, "Sun Network File Protocol Design Considerations," Internal Sun Microsystems technical note, January 1984. *A description of the basic design principles in the NFS protocol, rationale and implementation, including the use of a reply cache for correctness.*

[Joy84b] Joy, Bill, "Design of the Sun Network File System," Internal Sun Microsystems technical note, January 1984. *Design of the implementation, relationship to VFS, comparison to ND, omissions in design and reasons, and related work.*

[Juszczak89] Juszczak, Chet, “Improving the Performance and Correctness of an NFS Server,” *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1989, pages 53-63. *Describes a server reply cache implementation for work avoidance. Listed as a side-effect, the reply cache avoids the destructive re-application of nonidempotent operations—improving correctness.*

[Juszczak94] Juszczak, Chet, “Improving the Write Performance of an NFS Server” *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a write gathering technique that exploits NFS client implementation parallel write behavior to improve write throughput in NFS Version 2.*

[Kazar88] Kazar, Michael Leon, “Synchronization and Caching Issues in the Andrew File System,” *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Dallas Winter 1988, pages 27-36. *Describes cache consistency in AFS and contrasts it with other distributed file systems.*

[Kazar90] Kazar, Michael Leon, Leverett et al., “DEcorum File System Architectural Overview,” *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Anaheim June 1990. *Describes the DCE DFS file system.*

[Kazar94] Kazar, Michael, private communication April 1, 1994. *Mike is right—the reply cache is volatile state.*

[Macklem91] Macklem, Rick, “Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol,” *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes performance improvement (reduced CPU loading) through elimination of data copies in tuning the 4.3BSD Reno NFS implementation, and the performance and use of TCP as a transport.*

[Macklem94] Macklem, Rick, “Not Quite NFS, Soft Cache Consistency for NFS,” *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1994. *Describes a cache consistent NFS protocol, with extensions similar to the work described here.*

[McVoy91] McVoy, L., Kleiman, S., “Extent-like Performance from a UNIX File System,” *Winter USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, January 1991. *Describes a write clustering technique for UNIX local file system writes to improve throughput.*

[Mogul92] Mogul, Jeffrey C., “A Recovery Protocol for Spritely NFS,” *USENIX File System Workshop Proceedings*, Ann Arbor, MI, USENIX Association, Berkeley, CA, May 1992. *Second paper on Spritely NFS proposes a scheme for recovering state in a consistency protocol.*

[Mogul93] Mogul, Jeffrey C., “Recovery in Spritely NFS,” Research Report 93/2, Digital Equipment Corporation Western Research Laboratory, June 1993. *Third paper on Spritely NFS describes the implementation of recovery.*

[Moran90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., “Breaking Through the NFS Performance Barrier,” *Proceedings of the 1990 Spring European UNIX Users Group*, Munich, Germany, pages 199-206, April 1990. *Describes the application of nonvolatile RAM in solving the synchronous write bottleneck in NFS Version 2.*

[NFS3] Sun Microsystems, Inc., “NFS Version 3 Protocol Specification,” February 16, 1994.

[Nelson88a] Nelson, Michael N., Brent B. Welch and John K. Ousterhout, “Caching in the Sprite Network File System,” *ACM Transactions on Computer Systems* 6(1). February, 1988. Also Computing Reviews, Vol. 30, No. 3, March 1989. *Caching strategies, consistency protocol and performance results.*

[Nelson88b] Nelson, M.N., “Physical Memory Management in a Network Operating System,” *Ph.D. Thesis*. Univ. of Calif., Berkeley. November, 1988.

[Nowicki89] Nowicki, Bill, “Transport Issues in the Network File System,” ACM SIGCOMM newsletter *Computer Communication Review*, April 1989. *A brief description of the basis for the dynamic retransmission work.*

[Ousterhout90] Ousterhout, John K., “Why aren’t Operating Systems Getting Faster as Fast as Hardware,” Proceedings of the 1990 Summer USENIX Conference, Anaheim, June 11-15, 1990. *A description, in part, of the synchronous write bottleneck in NFS Version 2.*

[POSIX90] Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language] ISO/IEC 9945-1: 1990, IEEE Std 1003.1-1990.

[Pawlowski89] Pawlowski, Brian, Ron Hixon, Mark Stein, Joseph Tumminaro, “Network Computing in the UNIX and IBM Mainframe Environment,” *Uniforum ‘89 Conf. Proc.*, (1989). *Description of an NFS server implementation for IBM’s MVS operating system.*

[Presto93] Digital Equipment Corporation. “Guide to Prestoserve,” DEC OSF/1 Prestoserve Product Documentation, Order number AA-PQTOA-TE, March 1993.

[RFC1014] Sun Microsystems, Inc., “External Data Representation Specification,” RFC-11014, DDN Network Information Center, SRI International, Menlo Park, CA. *Describes canonical data exchange format for use with RPC.*

[RFC1057] Sun Microsystems, Inc., “Remote Procedure Call Specification,” RFC-1057, DDN Network Information Center, SRI International, Menlo Park, CA.

[RFC1094] Sun Microsystems, Inc., “Network Filesystem Specification,” RFC-1094, DDN Network Information Center, SRI International, Menlo Park, CA. *NFS Version 2 protocol specification.*

[Reid90] Reid, Jim, “N(e)FS: the Protocol is the Problem,” *Proc. of the UKUUG Conference*, London, July 1990. *Describes problems in NFS Version 2.*

[Sandberg85] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, “Design and Implementation of the Sun Network Filesystem,” *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, Summer 1985. *The basic paper describing the SunOS implementation of NFS Version 2.*

[Satyanarayanan89] Satyanarayanan, M., “A Survey of Distributed File Systems,” *Annual Review of Computer Science*, Annual Reviews, Inc. Volume 4, 1989. Also available as Technical Report CMU-CS-89-116, Dept. of Comp. Sci., Carnegie Mellon University. *A survey of NFS, AFS and other distributed file systems with a comprehensive bibliography.*

[Srinivasan89] Srinivasan, V., Jeffrey C. Mogul, “Spritely NFS: Implementation and Performance of Cache Consistency Protocols”, WRL Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, 100 Hamilton Ave., Palo Alto, CA, 94301, May 1989. Also in Proc. of the Twelfth ACM Symposium on Operating Systems Principals. *Analysis of a Sprite-like consistency protocol applied to NFS.*

[Welch90] Welch, Brent, "Measured Performance of Caching in the Sprite Network File System" *Computing Systems*, Volume 3, Number 4, Summer 1991, pages 315-342. *Analyzes the effectiveness of caching in the Sprite network file system, and the frequency of concurrent write sharing.*

[Wittle93] Wittle, Mark, Bruce Keith, "LADDIS: The Next Generation in NFS File Server Benchmarking" *Proc. Summer 1993 USENIX Conference*, USENIX Association, Cincinnati, OH, June 1993, pages 111-128. *Describes a synthetic, parallel benchmark used to evaluate NFS Version 2 server performance.*

[X/OpenNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: XNFS, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991. *Describes the NFS version 2 protocol and accompanying protocols, including the Lock Manager and the Portmapper.*

[X/OpenPCNFS] X/Open Company, Ltd., X/Open CAE Specification: Protocols for X/Open Internetworking: (PC)NFS, Developer's Specification, X/Open Company, Ltd., Apex Plaza, Forbury Road, Reading Berkshire, RG1 1AX, United Kingdom, 1991.

Author information

Chet Juszczak is a Consultant Engineer in the UNIX Software Group at Digital where he works on distributed file systems for commercial servers. He has worked on NFS and file server performance at Digital since 1985. Before that he worked on relational database technologies at AT&T. Chet got his M.S. in C.S. at the University of Michigan in 1983. Reach him electronically at chet@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua, NH 03062.

Brian Pawlowski now resides at Network Appliance Corp., where he works on performance and new architectures for appliances. Brian did the work described here while at Sun Microsystems, Inc., where for six years he worked on distributed file systems. He led the MVS/NFS project, slummed with AFS and DCE DFS, worked on LADDIS and finally became aware while watching the pretty colors during a multiprocessor file server performance project that provided a convenient excuse to use some sophisticated performance visualization tools. Brian has never castrated nor slaughtered cattle. Reach him electronically at beepy@netapp.com or via snail mail at Network Appliance, 295 N. Bernardo Ave., Mountain View, CA 94043.

Peter Staubach is a staff engineer at SunSoft Inc. He is the project leader, principal designer and engineer on the NFS Version 3 and NFS over TCP projects. Peter has been involved with the design and implementation of SunSoft's distributed file system since 1991. Prior to joining Sun, Peter worked at Lachman Associates where he was involved with the implementation of NFS for System V and sundry NFS ports. He can be reached electronically at staubach@eng.sun.com or via U.S. Mail at SunSoft Inc. 2550 Garcia Ave. MS MTV05-40, Mountain View, CA 94043.

Carl Smith has been a member of the technical staff at Sun for six years. During this time he has been the NFS project lead, NFS Version 3 co-conspirator, and worked on various aspects of NFS, RPC, and TCP/IP. His friends have observed that he is working his way down the protocol stack. Prior to Sun he was a jack-of-all-trades at UniSoft Corporation and the technical manager and a principal engineer of the Berkeley PDP-11 Software Distribution. He has been involved with UNIX since Version 6. His current interests are networking, security, and retiring early to visit the bookstores of the world. He can be reached electronically at cs@eng.sun.com or via U.S. Mail at SunSoft Inc. 2550 Garcia Ave. MS MTV05-44, Mountain View, CA 94043

Diane Lebel is an engineer in the UNIX Software Group at Digital where she is the principal designer of the DEC OSF/1 NFS client. Diane has worked at Digital since 1987. Reach her electronically at lebel@zk3.dec.com or via U.S. Mail at Digital Equipment Corp., 110 Spit Brook Rd., Nashua, NH 03062.

David Hitz is a co-founder and director of system architecture at Network Appliance Corp. Dave has focused on designing and implementing the WAFL file system, and on the overall design of their file server. He also worked at Auspex Systems in the file system group, and at MIPS in the System V kernel group. He received his computer science BSE from Princeton University in 1986. Reach him electronically at hitz@netapp.com or via snail mail at Network Appliance, 295 N. Bernardo Ave., Mountain View, CA 94043.

Trademarks

NFS is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories, a wholly-owned subsidiary of Novell, Inc. Prestoserve is a trademark of Legato Systems, Inc.

Appendix 1. Summary of the NFS Version 3 protocol. Notes: fh = file handle, dir_fh = directory file handle, wcc_data = weak cache consistency data.

	Procedure	Arguments	Returns	Change from NFS Version 2
0	`NULL`—do no work	*none*	*none*	
1	`GETATTR`—Get attributes of a file	fh	attributes	File types are defined for sockets and named pipes as well as block and character devices, with explicit subfields for major and minor device numbers.
2	`SETATTR`—Set attributes of a file	fh, new attributes	wcc_data	Now uses a discriminated union for each field, telling whether the field is to be set (and how), eliminating unsigned field overloading with `-1`. The time fields can be set to either the server's current time or the time supplied by the client.
3	`LOOKUP` a file name	dir_fh, name	fh, attributes, directory attributes	Returns the attributes for the directory searched for more efficient name cache validation.
4	`ACCESS`—check access permissions	fh, access mask	attributes, access mask	New procedure, allows over-the-wire access check at open to detect permission failures. Also useful for ACL-based implementations.
5	`READLINK`—read from a symbolic link	fh	attributes, symlink data	Returns attributes for symbolic link.
6	`READ` data from a file	fh, offset, count	attributes, count, eof flag, data	Reply structure includes an `EOF` boolean allowing clients to determine efficiently the end of file.
7	`WRITE` data to a file	fh, offset, count, stable flag, data	count, committed flag, verifier, wcc_data	An enumerated field indicates disposition of request. Removed the `beginoffset` and `totalcount` fields in the NFS Version 2 protocol. The reply contains a count so that the server can write less than the requested amount and correctly indicate this to the client.
8	`CREATE` a file	dir_fh, name, attributes to apply	fh, attributes, wcc_data	Contains an exclusive flag to support exclusive creation of regular files (in conjunction with an exclusive create verifier). Returns wcc_data for directory.
9	`MKDIR`—create a directory	dir_fh, name, attributes to apply	fh, attributes, wcc_data	Returns wcc_data for parent directory.
10	`SYMLINK`—Create a symbolic link	dir_fh, name, attributes to apply, data	fh, attributes, wcc_data	Returns wcc_data for directory and a file handle and attributes for the symlink.
11	`MKNOD`—Create a special device	dir_fh, name, attributes to apply, special data	fh, attributes, wcc_data	New procedure allows creation of special files without overloading the fields in the `CREATE` procedure.
12	`REMOVE` a file	dir_fh, name	wcc_data	Returns wcc_data for directory.
13	`RMDIR`—remove a directory	dir_fh, name	wcc_data	Returns wcc_data for parent directory.
14	`RENAME` a file	source dir_fh, source name, target dir_fh, target name	wcc_data, wcc_data	Returns wcc_data for both directories.
15	`LINK`—create a hard link	fh, target dir_fh, name	attributes, wcc_data	Returns wcc_data for directory
16	`READDIR`—read directory entries	fh, cookie, verifier, count	directory attributes, verifier, entries	Arguments include a verifier to allow the server to validate the directory offset. Returns attributes for directory
17	`READDIRPLUS`—Read directory entries/attributes	fh, cookie, verifier, count, maximum size	directory attributes, verifier, entries	New procedure returns file handle and attributes in addition to directory entries.
18	`FSSTAT`—volatile file system information	fh	attributes, file system utilization	Reply includes total size and free space in a file system in bytes (removing the ambiguous notion of blocks as the file system size), total number of files and free file slots, and an estimate of the time between file system modifications (for cache consistency checking algorithms).
19	`FSINFO`—Get static file system information	fh	attributes, file system properties	Provides nonvolatile file system information including preferred/maximum read/write transfer size, and flags stating whether links or symbolic links are supported. Also includes preferred transfer sizes for `READDIR` and `READDIRPLUS`, the server time granularity, and an indication of whether times may be set in a `SETATTR` request.
20	`PATHCONF`	fh	attributes, POSIX information	Retrieve POSIX info—maximum name length, number links, etc. supported by remote file system. This procedures has been removed from the MOUNT protocol and put in NFS where it belongs
21	`COMMIT` data on server to stable storage	fh, offset, count	wcc_data, verifier	Provides the synchronization mechanism for asynchronous writes.