The Role of Distributed State

John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Distributed state offers the potential for improving the performance, coherency, and reliability of distributed systems. Unfortunately, distributed state also introduces consistency problems, crash sensitivity, time and space overheads, and complexity; these problems make it difficult to achieve the potential benefits. This paper describes the advantages and disadvantages of distributed state, and presents the NFS and Sprite file systems as examples of different tradeoffs. It does not appear possible to achieve all the advantages of distributed state and also avoid all the problems; rather, system designers must make compromises based on the needs of their individual environments.

1. Introduction

Webster's New World Dictionary defines state as ''a set of circumstances or attributes characterizing a person or thing at a given time'' [4]. State plays a fundamental role in all computer systems. One way of characterizing computation is as a set of operations applied to an initial state in order to produce some (presumably more interesting) final state. In this interpretation, programming is the act of invoking and organizing state transitions. ''State'' includes all of the observable properties of a program and its environment, including instructions, variables, files, and input and output devices. Without state there would be no computers or computation; many of the recent advances in computer science have occurred because state (in the form of main memory and disk storage) has become cheaper and more plentiful.

In a distributed system, such as a network of workstations and servers, the overall state of the system is partitioned among several machines. The machines execute concurrently and mostly independently, each with immediate access to only a piece of the overall state. To access remote state, such as a memory location or device on a different machine, the requesting (or ''client'') machine must send a message to the machine that contains the state (called the ''server'' for the request). Many of the interesting issues in distributed systems stem from two properties of their state: first, some state must be accessed in a different fashion than other state; and second, if one machine in the system crashes, it causes some *but not all* of the overall state to be lost.

For this paper I will focus on *distributed state*, which I define loosely as ''information retained in one place that describes something, or is determined by something, somewhere else in the system.'' Some examples of distributed state are:

- A small table kept on each host to associate network addresses with the textual names of other hosts.
- A sequence number kept on a host to identify the most recent byte of data received from some other host.
- A block of a file cached in the main memory of one host even though the file is stored on a disk attached to a different machine.
- A table kept on a file server to keep track of the workstations that are caching a particular file.

Only a small fraction of all the state in a distributed system is distributed state: information that describes something on one machine and is only used on that machine (e.g. saved registers for an idle process) is not distributed state, by my definition.

The act of building a distributed system consists of making tradeoffs among various alternatives for managing the distributed state. This paper is a discussion of some of the alternatives and their implications. Section 2 describes the potential benefits offered by distributed state, and Section 3 then shows why it is difficult in practice to achieve the benefits. Sections 4 and 5 use two network file systems as case studies to illustrate the tradeoffs in managing distributed state. Finally, Section 6 concludes with the opinion that there is no perfect solution to managing distributed state: each system designer must choose a particular approach (which will necessarily have both advantages and disadvantages) based on the needs of his or her particular environment.

2. Why Is Distributed State Good?

Distributed state can be used to provide three benefits in a distributed system: performance, coherency, and reliability. Distributed state improves performance because it makes information available immediately; there is no need to send a message to a remote machine to retrieve the information. For example, a local table containing name-to-address mappings makes it unnecessary to contact a central name server each time a textual name must be mapped to its corresponding address. Or, if a machine caches a remote file in its main memory then the file can be read without re-reading the file from disk and potentially without even contacting the server to which the disk is attached.

The second potential advantage of distributed state is coherency. For machines (or people) to work together effectively, they must agree on common goals and coordinate their actions. This requires each party to know something about the other. For example, if a host keeps a sequence number identifying the most recent byte of data it received from some other host, and if each arriving packet contains a sequence number identifying the first byte of data in that packet, then the receiver can compare sequence numbers to detect when packets are duplicated or arrive out-of-order. Without the sequence number there would be no way to detect these common error conditions, and it would be much more difficult for machines to communicate. Another example is the one from above, where a file server keeps a table of file usage: if one workstation is about to write a file that is cached on several other workstations, the file server can notify the other workstations so that they don't use ''stale'' data from their caches.

The third potential advantage of distributed state is reliability. If a particular piece of information is replicated at several sites in a distributed system and one of the copies is lost due to a failure, then it may be possible to use one of the other copies to recover the lost information. For example, if a file server crashes but a workstation has one of its files cached, it might be possible for the workstation to make the file available to the rest of the system while the server reboots; after the server has rebooted it could reclaim jurisdiction over the file.

3. Why Is Distributed State Bad?

Unfortunately, the benefits of distributed state listed above are only *potential* benefits; in practice they are difficult to achieve. The following paragraphs describe four problems introduced by distributed state: consistency, crash sensitivity, time and space overheads, and complexity.

3.1. Consistency

The first problem with distributed state is consistency: if the same piece of information is stored at several places and one of the copies changes, what happens to the other copies? If the other copies are not updated then incorrect decisions may be made with the out-of-date information. Even if the other copies are eventually updated, there will be a window of time when the copies are inconsistent and this could cause the system to behave incorrectly. Approaches to the consistency problem fall into three classes:

Detect stale data on use. In some situations it is easy to detect attempts to use out-of-date information. In these cases, there is no need to update all the copies when one changes. If an attempt is made to use stale information, its staleness will be noticed and a fresh copy of the information can be fetched. As pointed out by Lampson [6], the name-to-address map is an example of this approach. Suppose that there is a change in the address corresponding to a given name. If each message contains the name of the desired host as well as its address, then message recipients can verify that each incoming packet has the correct name. If a machine attempts to use an out-of-date address this fact will be detected (either as a timeout or as an error return from the machine with the erroneous address), at which point the sender can contact a central name server to refresh its name-to-address mapping. This form of distributed state is sometimes called *hints* to reflect the fact that it need not always be correct.

Prevent inconsistency. The second approach is to mask the window of inconsistency, either by eliminating all but one copy of the information before each modification, or by preventing access to the out-of-date copies until they are updated. For example, in the Sprite system if a file is being modified by one workstation while being read by another workstation, then neither is allowed to cache the file; all read and write operations are passed through to the server and applied to its single copy of the file [8]. In Locus, it is possible for a file to be replicated on different disks attached to different servers [12]. If the file is modified, the changes are applied initially to a single copy of the file and then propagated to the other copies. During the propagation period all accesses to the file are directed to the one up-to-date copy.

Tolerate inconsistency. In some situations the errors caused by stale state information may not do any harm, so they can be tolerated during a brief period while the copies are updated. For example, in a distributed game it may be acceptable for there to be slight delays between when one player moves and another player perceives that move. Another example is the Grapevine mail system, where it can take several minutes for certain changes in configuration (such as the addition of a new user) to become visible everywhere in the system [2]. In general, users have a very low tolerance for inconsistencies of any sort (they tend to complain that the system is broken); fortunately, in the Grapevine case the inconsistencies are almost never noticed by anyone except system administrators.

3.2. Crash Sensitivity

The second problem with distributed state is crash sensitivity. In principle, distributed state should enhance the reliability of a system: if one machine fails then another should be able to take over its function. However, this only works if the replacement machine can reconstruct the state that had existed on the failed machine at the time of its failure. If the replacement machine cannot recreate the exact state of the failed machine then it will not be able to take over in a seamless fashion and the failure of the primary machine will be visible to other parts of the system.

In practice it is rare for state to be fully replicated (but see [2,12,14] for examples where it is). More commonly, each of the several distributed components has a different piece of the overall state, so that the failure of any component makes the entire system unusable. This sort of a distributed system is less reliable than a centralized system with only one component to fail. Most network file systems fall into this category: users invariably manage to spread essential files across all of the system's file servers, so that no-one can get any work done if any file server is down. In the worst case, the entire system has to be re-initialized when any component fails. In a slightly better scenario a crash on one file server ''only'' prevents people from working while the machine is down; activity will resume normally (without the need to restart other machines or programs) when the failed machine reboots.

If state is to be fully replicated in order to mask failures, several difficult problems must be resolved, including the following:

- The communication protocols must be designed in a way that redirects message traffic to the replacement machine after the failure of the primary machine.
- A failure may occur during the window of inconsistency when one replica has been modified but the others have not yet been modified. The communication protocols must be able to determine which copies have been updated, and the out-of-date copies must be brought back into consistency without waiting for the failed machine to reboot.
- When the failed machine eventually restarts, it must be able to use the replicas to bring its state into consistency with its replicas (the state could have changed substantially while the machine was down). In some cases the revived machine may be able to collect a complete snapshot of the state from a replica. In other cases (e.g. where the state involved is a large replicated file system) it may be too expensive to copy the entire state to the reviving machine; the backup machine may have to keep a record of changes and ''replay'' them for the reviving machine. Finally, the catching-up of the reviving machine must be synchronized with its participation in new requests made by clients.

All of these problems are solvable, but the solutions tend to be complex or inefficient, particularly if they are implemented in a general-purpose fashion. Some of the most successful approaches use information about a particular problem domain to implement replication for that domain. See [1,2,3,12,14] for examples of the use of replication.

3.3. Time and Space Overheads

The third problem with distributed state is that it introduces overheads, both in time and in space. The time overheads are incurred mainly in maintaining consistency. Either the consistency of distributed state must be checked every time the state is used (for example, by contacting a file server to see if a cached copy of a file contains the most recent version), or some party must keep track of the distributed copies and notify each owner of a copy when the state changes (for example, the file servers in AFS perform this function [5]). If replicated copies are to be kept up-to-date then each update must be reflected in each of the copies. This overhead can make replicated updates substantially more expensive than non-replicated ones.

The most obvious source of space overhead is the storage needed for distributed copies of the same state (e.g. a single file may be cached on many workstations). However, there may be other space overheads to keep track of the distributed copies so that they can be kept consistent. In some environments, such as the Sprite file system, the space required for consistency-related information can be substantial.

The overhead problems are closely related to the degree of sharing and rate of modification. If information is not widely shared, then there need not be many copies of the information and it will not take much time to keep them all consistent. If there are many copies, then the space overhead increases. If shared information is updated frequently, then consistency actions will be invoked more frequently. At some point the cost of maintaining consistency becomes higher than the cost of communicating with a central server on each use; when this occurs, performance can be improved (and the system can probably be simplified) by reverting to a centralized approach to state management.

3.4. Complexity

The final problem with distributed state is complexity. Without distributed state there is no need to deal with consistency (it isn't a problem), nor is there any possibility of masking failures (one of the best things about a centralized system is that the whole system stops whenever any component stops). Distributed state makes a system substantially more complicated. The complexity makes it harder to debug the system and thereby reduces the reliability advantages offered by distributed state. Complexity also makes it harder to tune the system's performance (system implementors spend more time ‘‘getting it right’’ and less time ‘‘making it fast’’), thereby reducing the performance advantage offered by the distributed state.

4. Case Study #1: The NFS File System

To illustrate some of the issues in managing distributed state, this section and the next describe two network file systems: NFS and Sprite. NFS is a commercial product; it was originally developed by Sun Microsystems but it has become a *de facto* standard supported by almost all workstation vendors [13]. The Sprite file system was developed in a research project at the University of California at Berkeley [8,10]. Both systems use a client-server model with caching, as shown in Figure 1:

files are stored on disks attached to server machines, and clients make requests of the servers in order to access the files. Each system defines a particular set of possible requests, which represents a particular set of tradeoffs among the advantages and disadvantages inherent in distributed state. As a consequence, each system ended up with a corresponding set of good and bad properties; the strengths and weaknesses of the two systems are almost opposites.

The NFS design was optimized for simplicity and robustness, with performance a secondary goal. Simplicity and robustness were achieved by using a *stateless* protocol with *idempotent* operations. The term ''stateless'' means that file servers need not retain any information in their main memories. All essential information about the file system, such as the contents of files, must be kept on disk. As part of servicing each client request, the server must write any modified information to disk, so that future requests can be serviced even if the contents of the server's memory are lost (in a server crash, for example). Servers may cache disk blocks and other information in their main memories to improve performance, but the system must not depend on this information to function correctly. The term ''stateless'' is something of a misnomer, in that (a) it only applies to servers, (b) it only applies to the servers' main memories, and (c) it permits state in the main memories as long as that state is also on disk.

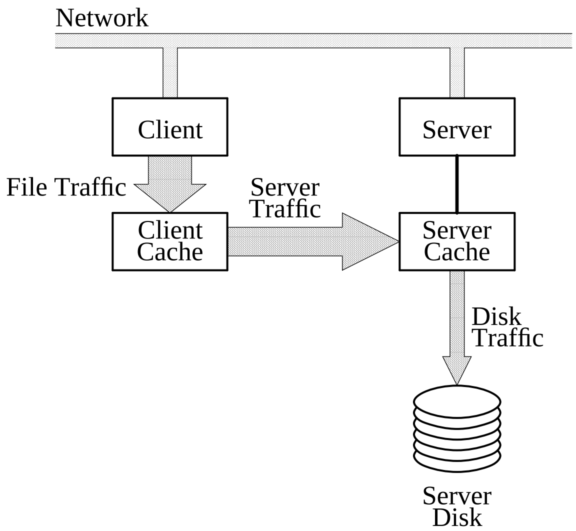

Figure 1. A network file system. File blocks are cached in the main memories of servers (machines with disks) and clients (machines without disks). When a process on a client machine attempts to read a file (''File Traffic''), the information is retrieved from the client's cache if it is present. If the desired information is not in the client's cache, the client issues a network request to the file server (''Server Traffic''); the server retrieves the information from its own cache (reading it from disk if it wasn't already in the cache) and returns the information to the client.

The second important characteristic of NFS is that almost all of its operations are idempotent. An idempotent operation is one that can be executed many times with the same overall effect as if it were executed only once. Some examples of idempotent NFS operations are:

read(fileId, position, count): given an identifier for a file and a byte position within that file, return **count** bytes starting at that position. Note that this operation would not be idempotent if the position were not specified by the client as part of the operation but instead were kept on the server and incremented by **count** as part of each read request.

write(fileId, position, count, data): given an identifier for a file and a byte position within that file, replace **count** bytes with data supplied by the client.

lookup(fileId, name): given an identifier for a file (which must be a directory) see if a particular name exists as an entry in that directory. If so, return the identifier for that file and its attributes (which include last-modified time, permissions, size, etc.). The identifier may be used by the client in later operations such as **read** and **write**.

Although almost all of the NFS operations are idempotent, there are a few non-idempotent operations, such as:

mkdir(fileId, name, attr): given an identifier for a directory, create a subdirectory in that directory with a given name and attributes. This operation is not idempotent because it returns an error if the given name is already present in the directory. Invoking it multiple times will result in success on the first invocation and failure on the second and later invocations.

In NFS, distributed state is kept almost exclusively on the clients. Servers do not store any information about their clients except for a list indicating which clients are allowed to access which disk partitions. Servers need not keep track of which files are currently in use or which clients are using which files. In fact, NFS servers *cannot* keep track of this information: there are no ‘‘open’’ or ‘‘close’’ requests to indicate when clients start and stop using files. NFS clients do keep distributed state, however. This state includes the following:

[1] File identifiers, returned by the **lookup** request and used in other requests, such as **read**.

[2] File data, returned by **read** operations and cached on clients to eliminate server requests if the information is re-used.

[3] File attributes, returned by **lookup** and other requests.

[4] Name translations (results of recent calls to **lookup**), cached on clients in order to bypass future calls to **lookup** for the same **fileId**s and **name**s.

Of this information, only the file identifiers are necessary for the system to function; the other information is cached in order to reduce the number of calls to file servers.

4.1. Advantages of NFS

Perhaps the greatest advantage of the NFS protocol is the ease with which it handles server crashes. If a server crashes, client requests will not be answered; the clients will detect the timeouts and simply retry their requests until eventually the server reboots and the requests succeed. Clients need not take any special action to handle server crashes since the retry mechanism is already required to handle lost packets. All important server state is on disk so nothing is lost during the crash (unless the disk was corrupted, which rarely happens). If the server crashes after completing an operation but before sending a response back to the client, then the client will re-issue the request after the server reboots, but this causes no problem for idempotent operations. To users on client machines, the server crash appears as a delay during which some processes are suspended, but when the server reboots all existing processes continue normally and seamlessly.

In NFS there is not enough replicated state to permit access to a server's files while it is down, and this is a disadvantage (which, by the way, is shared by almost all network file systems; the best-known counter-example is Locus [12]). But NFS has the important property of allowing client machines to survive server crashes without rebooting; the crash results only in delay, not in loss of state.

Another advantage of the NFS protocol is its simplicity, which stems directly from the stateless nature of the protocol. For example, crash recovery is handled without special code on either the client or server. The stateless protocol results in a small set of operations with simple interactions between clients and servers; this makes it easy to build NFS clients and servers. Although most implementations of NFS have been made in variants of the UNIX operating system, there are also exist NFS implementations for other systems, such as MS-DOS for the IBM PC.

4.2. Disadvantages of NFS

Unfortunately, statelessness is also a source of problems in NFS. The NFS protocol suffers from three major weaknesses: performance, consistency, and semantics. The greatest problem with NFS is its performance, which is limited by the stateless protocol. Whenever a client issues a write request, the server must guarantee that all modified data is safely on disk before the write returns. Not only must the file's data be written to disk, but the file's descriptor must also be flushed along with any index blocks that have changed. This results in two or three disk transfers for every block of file data. When a large file is written, each block of the file will result in a separate write request, so the file's descriptor and index blocks will be written to disk over and over. As a result, NFS clients cannot typically achieve write bandwidths greater than about 60 Kbytes/sec. In contrast, UNIX systems with local disks can usually achieve write bandwidths of 500-1000 Kbytes/sec.

Non-volatile memory may make it possible to alleviate some of the performance problems caused by statelessness. For example, Legato Systems offers an NFS accelerator that uses a small non-volatile memory unit as a write buffer for the disk. The cache has a much faster access time than the disk. When descriptors and index blocks are repeatedly written, as described above, the writes are made to the non-volatile memory. Only a single disk write will be necessary when the information ages out of the cache. Because the cache is non-volatile, it can survive server reboots just as well as the disk. However, non-volatile memory does not eliminate all of the performance problems with NFS: NFS still requires extra I/O operations to the cache, and it also requires additional server traffic as described below.

The second problem with NFS is consistency. Consistency problems arise because servers do not keep track of which clients are using which files. If one client modifies a file, there is no way for the server to notify other clients that have cached the old contents of the file; it is up to the other clients to find out on their own. This is achieved by polling. Whenever a file is accessed on a client, the client checks to see how recently the attributes for the file were fetched from the server. If the attributes are more than a few seconds old, the client refetches them. If the last-modified-time in the new attributes does not match the last-modified-time in the client's old copy of the attributes, then the client invalidates its cached data for the file. Similarly, cached name translations are also invalidated when they become more than a few seconds old. This approach ensures that each client eventually receives up-to-date information, but it permits windows of inconsistency where stale data may be used. Because of this, NFS cannot be used for certain applications where consistency is required, and it occasionally produces counter-intuitive behavior.

The consistency issue also impacts the performance of NFS systems. For example, the polling approach described in the previous paragraph results in extra server traffic. Even worse, NFS uses a *write-through-on-close* policy to reduce windows of inconsistency. Whenever a file is closed on a client machine, the client immediately transmits modified data for the file back to the server. The close operation does not complete until the data is safely on the server's disk. This approach is necessary in order to make the file's new data available to other clients as quickly as possible; if the new data is not returned to the server, then other clients will have no way of knowing that the file has changed.

Write-through-on-close has two unpleasant consequences. First, it delays the closing process until the data is written to disk. Second, it results in unnecessary load on the server and the disk. Many files are deleted or overwritten shortly after they are created [9]; if new data were retained for a while on the client before transmitting it to the server, much of the new data would be deleted and would never need to be transferred to the server or disk at all. Unfortunately, the statelessness of NFS requires that new data be returned immediately to the server to reduce consistency problems, and the only way to return data to the server is with the write operation, which forces data to disk.

The third problem with the NFS approach is that it introduces semantic difficulties. Statelessness and idempotency impose constraints that make it impossible or expensive to implement certain features. When a conflict occurs between a particular feature and statelessness or idempotency, there are two choices: don't implement the feature, or violate the goals of statelessness and idempotency. NFS uses both approaches. For example, file locking requires the server to keep track of which files are locked; the stateless model prohibits servers from keeping lock information solely in memory. Locks could have been implemented by writing the lock information to disk, but that would have made locking slow. The NFS designers decided not to provide locking at all (it was later provided by a separate network service). Another example of a semantic conflict is the **mkdir** operation described above. This operation is non-idempotent by definition, but had to be included in the protocol anyway. As a result, server crashes (or even lost packets) can produce unexpected behavior. For example, a **mkdir** could be processed by the server successfully, but if the response packet is lost the client will retry; the retry will fail because the file now exists.

4.3. NFS Summary

The best features of NFS are its simplicity and robustness; because of them, NFS is an overwhelming commercial success and a *de facto* standard. The semantic difficulties in NFS do not arise often in practice (for example, sharing of a single file within a period of a few seconds is uncommon, so the windows of inconsistency are not usually noticed). Even NFS's performance problems have not been a problem on slower workstations that are limited more by CPU speed than disk speed. However, newer workstations with CPU speeds of 10 MIPS or more are severely hampered by NFS's ‘‘flush-to-disk’’ approach. It seems likely that changes will have to be made in the NFS protocol to improve its performance for the even faster workstations of the future.

5. Case Study #2: The Sprite File System

The Sprite file system appears on the surface much like NFS. It uses a client-server model, clients use a request-response protocol to communicate with the servers, the actual requests bear quite a bit of surface similarity to those in NFS, and clients keep many of the same kinds of distributed state as in NFS. However, Sprite manages the distributed state of the file system in a different fashion than NFS. The result is a system with almost totally opposite strengths and weaknesses: Sprite provides high performance and clean semantics, but it is more complex and faces more difficult crash recovery problems.

The protocol between clients and servers is definitely not stateless in Sprite; we call it ‘‘stateful’’ for lack of a better term. Three additional pieces of distributed state are kept in Sprite:

[1] Servers keep information in their main memories about which workstations are reading or writing which files. This requires clients to notify servers whenever files are opened or closed, but allows the servers to enforce consistency as described below.

[2] Servers retain modified file blocks in their main memories, and do not write that information back to disk until it has aged for thirty seconds.

[3] Clients also retain modified file blocks in their main memories; they do not pass new information back to servers until it has aged for thirty seconds or until the

information is needed by some other client. If a client has dirty blocks for a file, the server's state information reflects this.

In contrast to NFS, Sprite does not keep name translation information on clients. In order for servers to maintain the state described above, clients must already contact servers whenever they open or close a file; in Sprite, the clients pass the entire multi-level file name to the server and let the server handle the name lookup.

5.1. Advantages of Sprite

By retaining additional state, the Sprite file system provides substantially better consistency and performance than NFS. Consistency is improved because Sprite file servers can use their state information to prevent stale data from being used. If a file is ever open simultaneously on several clients and at least one of them is writing the file, then the server notifies each of the clients and insists that they not cache the file; all read and write operations must be passed through to the server, where they are applied to a single copy of the file in the server's cache. If a client has a cached copy of a file, but the file isn't open on that client, then the client will not be notified if other clients modify the file; stale data will remain in its cache. However, that data cannot be used until the file is opened. When the client makes an open request to the server, the server returns a version number for the file. This version number will not match the version associated with the stale data, so the file will be purged from the client's cache. Thus Sprite provides ‘‘perfect’’ file consistency: each read operation is guaranteed to return the most recently written data for that file, regardless of where and when the file is read and written.

Sprite's stateful approach also allowed file locking to be implemented easily, using the main memory of the server to record who owns which locks. Overall, the behavior of the Sprite file system as seen by users is identical to the behavior of a file system running on a single timeshared UNIX machine.

The second advantage of the Sprite file system, performance, is even more noticeable. Much of Sprite's performance is due to the way it handles consistency. Since the servers keep track of which clients are using which files, clients need not return modified file data to servers immediately. If some other client opens the file, then the server will retrieve the dirty data from the client that modified it. When servers eventually do receive information from clients, they do not force it immediately to disk, nor do they write the file descriptor and index blocks to disk every time a data block is written to disk. If the new data survives for thirty seconds, then it will be written to disk, and the corresponding file descriptor and index blocks will be written to disk *once*.

Sprite's approach has two performance advantages. First, clients need not wait for information to be written to disk when they close files. They can continue processing immediately; if the data eventually needs to be passed back to the server, a background kernel process does it. The second advantage is that some new data is deleted or overwritten before being passed back to the server, so the overhead of communicating with the server is never incurred. Measurements of our Sprite

Machine	Sprite (secs.)	NFS (secs.)	NFS Slowdown (%)
Sun-3/75	439	635	44
Sun-4/280	184	270	46
DECstation 3100	127	269	111

Table I. A performance comparison of the NFS and Sprite file systems on a modified version of the Andrew benchmark devised by M. Satyanarayanan [5]. The ‘‘Sprite’’ column gives the elapsed time for a diskless client to complete the benchmark when both the client and server machines were running Sprite. The ‘‘NFS’’ column gives the elapsed time when both the client and server ran a vendor-supplied version of Ultrix or SunOS with all (or almost all) file accesses made remotely using NFS. The ‘‘Slowdown’’ column indicates how much slower NFS was than Sprite. In each case the server machine was the same type as the client machine. See [11] for details of the benchmarking.

network indicate that only about 50% of newly written data is ever returned to the file server [15]. Table I compares the performance of Sprite and UNIX/NFS for a file-intensive benchmark. On identical hardware configurations, the benchmark ran 45% to 110% slower under UNIX/NFS than under Sprite. Mike Nelson’s dissertation shows that most of the performance difference is due to the difference in writing policy between the two systems [7].

5.2. Disadvantages of Sprite

Unfortunately, the stateful approach used in Sprite has disadvantages as well as advantages. The paragraphs below discuss four problems we had to face in Sprite: complexity, recovery, performance, and space overhead. First, Sprite’s file system is more complex than NFS. Most of the complexity is associated with managing the server’s state. The initial implementation suffered from subtle race conditions (see Figure 2 for an example) and network-wide deadlocks involving several clients and servers. Although we have gradually eliminated these problems, it has been difficult both to isolate the problems and to find simple solutions for them.

The second, and greatest, problem with the Sprite approach is recovery. Quite a bit of volatile information is kept in the main memories of clients and servers, and all of this information can potentially be lost in a crash. Fortunately, when Sprite machines crash they attempt to flush their file caches (to disk in the case of servers; to servers in the case of clients). This approach almost always works except for power failures, so file data is almost never lost (and when information is lost, it is confined to information written in the last minute).

Unfortunately, when a server reboots it loses its information about file usage. Thus it no longer knows which clients are using which files, so it may not be able to enforce consistency for files that are already open. In the original version of Sprite, all files open at the time of a server crash were forcibly closed when the server rebooted. This caused all work in progress to be lost, including shells and window systems; most users found it easiest to reboot their workstations in order to restore their execution environment. We very quickly decided that this approach was

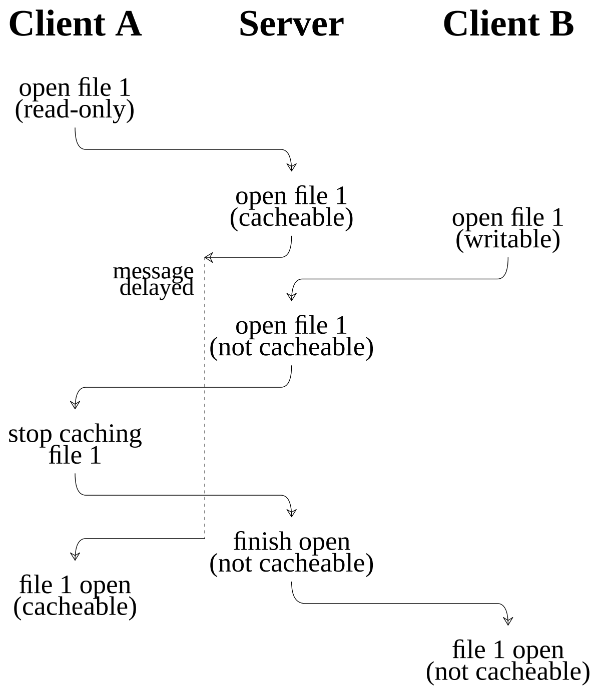

Figure 2. A race condition in managing cache coherency, which occurred in an early version of Sprite. Client A opens a file for reading; since A is the only client using the file, the server responds with an indication that the file is cacheable. Then client B opens the same file for writing. At this point the server decides that the file cannot be cached safely, and it sends a message to A indicating this fact. Unfortunately, the response to A's earlier open request may have been delayed (e.g. because the response packet was lost and had to be retransmitted after a timeout), so that the ''don't cache'' request arrives at A before the open response. When the open response eventually arrives at A, it causes A to cache the file, which is unsafe. The race condition was eliminated by keeping extra state on the client.

intolerable.

Fortunately, we discovered that distributed state was not just the cause of the recovery problem, but also the solution. By adding slightly to the state kept by clients about their files, it became possible for servers to recreate all their usage information after rebooting. A new operation was added to the Sprite protocol: **reopen**. When a server reboots, each client reopens all of its files by passing the server a copy of its state information (including information such as which files are open for reading or writing, which are locked, etc.). This allows the server to reconstruct its state so that it can continue to guarantee consistent access to files, and so that locks are not broken when servers reboot. Our experience is that this provides an effect almost identical to NFS: files are not accessible while a server is down, but when servers reboot the clients can continue operation without any loss of state.

Of course, Sprite's approach to recovery is more complicated than the NFS approach where there are no special recovery actions at all. It has also introduced another performance problem. Clients typically have several hundred files open on each of several servers. When a server reboots, all of the clients simultaneously attempt to re-open their files. Our current system contains about 40 machines, and the recovery traffic from these machines is so intense that we refer to it as a *recovery storm*. The recovery storms overload the servers to the point where they cannot respond to requests in a timely fashion. Some operations time out, causing clients to think the server has crashed, whereupon they re-initiate their file reopening from the beginning. As a result, most clients have to attempt recovery several times before successfully reopening all their files. If the system is to scale from its current size of 40 machines to 400, techniques will have to be found to deal with the contention induced by recovery. We are currently exploring a variety of approaches in the low-level communication protocol and in the file system. Although we expect to solve the problem, the solution will undoubtedly add more complexity.

A third potential problem with Sprite's approach is its requirement that each open and close be reflected through to the file's server. In contrast, NFS clients never contact servers during closes except to write new data; during opens, an NFS client need not contact the server as long as it has up-to-date attributes cached for the file. It would be possible to extend the Sprite mechanism so that clients cache naming information and need not contact the servers on every open or close; the Andrew file system already implements such a mechanism [5]. However, such a mechanism would add to the distributed state, thereby increasing the complexity of clients and servers (particularly because the cached naming information would have to be kept consistent). In addition, the performance benefit would be partially offset by server traffic to fill the clients' name caches and keep them consistent. Our performance measurements indicate that name caching would reduce server loading but would have relatively little effect on the performance of clients. Since server loading is not a major limitation in our environment we haven't implemented name caching (yet).

The last, and least important, problem with Sprite's approach is the space overhead for the servers' file state. A file server needs several hundred bytes of storage for each open file to keep track of the file's usage, and may have many thousand files open at once. As a result, the storage required for file state grossly exceeded our initial estimates, causing internal memory limits in the Sprite kernel to be exceeded. We solved the problem by increasing the internal limits, but we were surprised at how much space the state occupies. In typical configurations, a Sprite file server will use most of its memory (10-100 Mbytes) for caching file data, and the usage state information typically occupies about 15-20% as much space as the file data (many megabytes in larger configurations). Furthermore, we expect the usage information to increase as the number of client workstations increases. If the system increases in size by another order of magnitude, the size of the file usage information could potentially become a problem.

5.3. Sprite Summary

Our main goals in Sprite were to achieve high performance and timesharing semantics; those goals have been met. A secondary goal was to achieve reliability comparable to NFS. That goal has also been met, although it was not met in the initial version of the system. Unfortunately, meeting the goals has resulted in a system substantially more complicated than NFS and has introduced some scaling problems that have not yet been completely resolved.

6. Conclusions

I do not believe there is a ‘‘perfect’’ solution to the problems associated with distributed state. The simpler solutions, like NFS, are likely to have performance problems, and the faster solutions, like Sprite, tend to be more complicated and to present more difficult recovery problems. Systems that are more fault-tolerant tend to have even greater performance or complexity problems. In striving for some of the potential advantages of distributed state, system designers must necessarily embrace some of the disadvantages as well. The exact choice among the various options should reflect the needs of the environment being designed for: in some environments performance may be less important than the ability to survive machine failures, and vice versa.

Based on my experience with the NFS and Sprite file systems, I do not believe that the stateless model can meet the needs of high-performance workstations of the future. A stateless approach will limit the performance of the system to the performance of disks; unfortunately, disk performance is not improving at anywhere near the rate of processor performance. Non-volatile memory offers some hope for performance improvement, but I think the best solution is a change to more stateful protocols.

On the other hand, distributed state almost always introduces complexity and fragility, so system designers should attempt to reduce distributed state as much as possible. The less state, the better. In Sprite, I suspect that we may have been a little too eager to embrace state, and that a careful redesign of the system could reduce the amount of state we have to maintain.

Finally, the best approach to dealing with failures is to merge recovery with normal operation so that there is nothing special to do during recovery. NFS achieves this quite nicely through its combination of statelessness and idempotency. Recovery happens so infrequently that is is very difficult to debug special-case recovery code: it is hard to invoke the code under test conditions, and the code is hardly ever executed under real-life conditions. This means that there is a good chance that the code will not work when it is needed. On the other hand, if the recovery code and regular-case code are the same, the recovery code will be exercised constantly during everyday operation of the system so it is likely to work correctly when needed.

7. Acknowledgments

Mendel Rosenblum, Ken Shirriff, and Brent Welch made helpful comments that improved the presentation of this paper.

8. References

[1] Bartlett, J. ‘‘A NonStop Kernel.’’ Proceedings of the 8th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 15, No. 5, December, 1981, pp. 22-29.

[2] Birrell, Andrew D., et al. ‘‘Grapevine: An Exercise in Distributed Computing.’’ *Communications of the ACM*, Vol. 25, No. 4, April 1982, pp. 260-274.

[3] Borg, A., et al. ‘‘Fault Tolerance Under UNIX.’’ *ACM Transactions on Computer Systems*, Vol. 7, No. 1, February 1989, pp. 1-24.

[4] Guralnik, D., Editor in Chief. *Webster’s New World Dictionary*, Second College Edition, Simon & Schuster, 1982.

[5] Howard, J., et al. ‘‘Scale and Performance in a Distributed File System.’’ *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.

[6] Lampson, B. ‘‘Hints for Computer System Design.’’ Proceedings of the 10th Symposium on Operating Systems Principles, *Operating Systems Review*, Vol. 17, No. 5, December, 1985, pp. 33-48.

[7] Nelson, M. *Physical Memory Management in a Network Operating System*. Ph.D. Dissertation, technical report UCB/CSD 88/471, Computer Science Division, University of California at Berkeley, November 1988.

[8] Nelson, M., Welch, B., and Ousterhout, J. ‘‘Caching in the Sprite Network File System.’’ *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134-154.

[9] Ousterhout, J., et al. ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System.’’ *Proceedings of the Tenth Symposium on Operating Systems Principles*, December 1985, pp. 15-24.

[10] Ousterhout, J., et al. ‘‘The Sprite Network Operating System.’’ *IEEE Computer*, Vol. 21, No. 2, February 1988, pp. 23-36.

[11] Ousterhout, J. ‘‘Why Aren’t Operating Systems Getting Faster as Fast as Hardware?’’ Technical note TN-11, DEC Western Research Laboratory, October 1989.

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

[13] Sandberg, R., et al. ‘‘Design and Implementation of the Sun Network Filesystem.’’ *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.

[14] Satyanarayanan, M., et al. *Coda: A Highly Available File System for a Distributed Workstation Environment*. Technical report CMU-CS-89-165, School of Computer Science, Carnegie Mellon University, July 1989.

[15] Welch, B. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. Ph.D. Dissertation, University of California at Berkeley, 1990.