Synchronization and Caching Issues in the Andrew File System

Michael Leon Kazar

Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

ABSTRACT

Many distributed file systems go to great extremes to provide exactly the same consistency semantics in a distributed environment as they provide in the single machine case, often at great cost to performance. Other distributed file systems go to the other extreme, and provide good performance, but with extremely weak consistency guarantees. However, a good compromise can be achieved between these two views of distributed file system design. We have built one such system, the Andrew file system. It attempts to provide the best of both worlds, providing useful file system consistency guarantees along with good performance.

1. Introduction

This paper discusses two important parts of distributed file system architecture: consistency guarantees and performance. When we use the term **consistency guarantees** in the context of a distributed file system, we refer to that part of the file system's specification that describes the state of the file system as seen from one machine in the environment, given the state of the file system at other machines. As an example of a consistency guarantee, Sun Microsystem's NFS file system [Kleiman 86] guarantees that data written to a file by one machine can be read by any machine in the network environment as long as at least 3 seconds have passed after the write call has completed.

The performance of a distributed file system is harder to describe. In this paper, distributed file system performance is characterized by two quantities: the network traffic generated, and the processor load placed on the "bottleneck" machines, usually the file servers. We came to this informal definition of file system performance based on our concern with scaling our distributed file system. We are building a distributed file system intended to scale to thousands of concurrent users; performance bottlenecks in either the processors or the network itself would prevent smooth scaling.

Of course, there are other criteria for measuring file system performance; for example, one might be interested in the smallest possible latency between the arrival of a request and its handling. The statements made here do not directly apply to other definitions of performance. Nevertheless, many of our performance improvement techniques apply to several performance models.

The consistency guarantees provided by a distributed file system strongly affect the possible implementations for the file system. If one wants to build a high-performance distributed file system, one must begin with consistency semantics that admit a good implementation. Of course, file system consistency semantics also affects the usefulness of a distributed file system for building applications, especially for distributed applications. If a distributed file system provides consistency guarantees that are too weak to use in building a distributed application, then that application will not be able to use the distributed file system for storage and sharing, and the system will be a failure.

This paper discusses the relationship between a distributed file system's consistency semantics and its performance. It begins in Section 2 with a survey of several extant distributed file systems, with particular emphasis on both performance bottlenecks and file system consistency semantics.

The next few sections describe the Andrew file system, a file system that makes different consistency/performance trade-offs from those described previously. Section 3 describes our goals in building the Andrew file system, along with a high-level description of the facilities it provides. Section 4 describes the implementation of the file system, and Section 5 describes our consistency guarantees, and how we implement them efficiently in practice.

Finally, this paper concludes by describing our experiences actually using the Andrew file system, with particular regard to how well the goals enumerated in Section 3 were achieved in practice. The paper concludes that efficiency and file system semantics can be balanced, leading to both good system performance along with a powerful system that can be used for sharing data among many machines easily.

2. Other Distributed File Systems

With the increasing use of local-area networks for connecting workstations and mainframe computers together, distributed file systems have been sprouting like dandelions. This section describes several such file systems: Suns' NFS distributed file system, the Apollo Domain system, AT&T's Remote File Sharing file system, the Cambridge Distributed Computing system and UCLA's Locus file system.

The description present here is not complete; its primary emphasis is on describing the file systems from two points of view: the communications architecture, and the consistency guarantees provided. The communications architecture is crucial in estimating the system's performance. In many systems the communications costs, including operating system's overhead, easily dominate all others; for such systems, analyzing the communications architecture gives a good idea of where bottlenecks will appear in actual operation. Furthermore, the communications architecture includes the client/server interface for the file system. This interface helps determine when the file server machines will become bottlenecks.

2.1. The Cambridge Distributing Computing System

The Cambridge distributed computing system [Needham 82] provides file service through the Cambridge file server, an instance of what Birrell and Needham call a "Universal File Server" [Birrell 80].

The file server provides two types of objects, normal **data files**, and **index files**. Index files contain simple directories, and the file server understands their internal structure, at least as far as being able to locate references from an index file to other files. This allows the file server to check the consistency of the file system, ensuring that no index file contains a reference to a non-existent file, and that every file is contained in at least one index file.

The basic operations provided by the Cambridge file server are **open**, **read**, **write** and **close**. The *open* call maps a file name into a temporary identifier (called a **TUID**), which can be thought of as a capability for the file. The *read* and *write* operations read and write data to a file named via a TUID, and the *close* operation takes a TUID and performs finalization processing.

The Cambridge file server actually provides two types of consistency guarantees, one for special files, for which it provides some atomicity guarantees, and one for normal files.

For normal files, read and write operations occur instantly on the file server, giving a very simple, yet useful, consistency guarantee: after a *write* operation completes, a *read* operation anywhere in the distributed environment reads this latest data.

For special files, slightly different guarantees are given. *Writes* to a special file are processed atomically, at the time the *close* call is made. Before that time, none of the new data is visible to others; afterwards, all of the new data is visible. Optionally, after writing some or all of the data to the new file, the writer of this new data may elect to abort the operation, rather than close the file. In this case, the new data is discarded, and the file reverts to its state before the transaction started. These atomic transaction semantics are implemented via a shadow page mechanism.

All four of these requests, *open*, *close*, *read* and *write*, are all handled by the file server, and all result in network communication.

2.2. Sun's Network File System

The Sun's Network File System (NFS) [Kleiman 86] splits the Unix file system into two levels, the top being the conventional Unix file system interface provided by the Unix system calls, and the bottom being what Sun calls the **vnode** interface (for virtual **inode**). A vnode is intended to be a generalization of the Unix inode, that is, a low-level, efficiently accessed file identifier.

There are several types of vnodes: **files, directories** and **symbolic links.** The vnode interface defines operations on these objects.

A typical file system operation decomposes into several vnode-level operations, for example, translating a pathname into its corresponding vnode and applying some vnode-level operation to the resulting vnode. The pathname is evaluated by starting at the root and in turn searching each directory for the name of the next component. The directory search routine is part of the vnode interface; given a directory vnode and a name within that directory, it returns the vnode of the named file or directory. When finished with the entire pathname, the file system has evaluated the corresponding vnode.

Miscellaneous operations, such as *rename*, *delete* and *link*, are applied directly to affected vnodes, or their containing directories.

At first glance, this interface would seem to require a message to the file server for every component of every pathname evaluated. While this cost can be reduced significantly by the use of caching at the client end of the system, a new question arises: how does one ensure the cache contains current information?

NFS handles cache consistency using a simple heuristic: file data is assumed to be valid for 3 seconds after it is fetched from a server; directory data is assumed to be valid for 30 seconds after its arrival.

Such a simple approach has both advantages and disadvantages. On the plus side, NFS is a fairly efficient implementation of a distributed file system. NFS makes extensive use of caching, and this reduces considerably the load an NFS client puts on its file server. And the cache consistency algorithm is easy to implement.

There are two points on the minus side, however. The most significant disadvantage of this approach is that it provides consistency guarantees that are simply too weak for many applications. A program such as a distributed *make* program (a Unix system building tool) would be very difficult to implement under NFS, as the system would have to wait 3 (30 seconds if a directory change is involved), for results to propagate from one machine to another. Such delays could easily make up for any improvement gained from concurrent compilation. This point is discussed in more detail in Section 4.

Another disadvantage to the NFS consistency scheme is, surprisingly, that it still sends too many version-checking messages. Because clients may contact the server every 3 seconds for heavily used files, many messages may be exchanged communicating the fact that the client's cache is still valid. Thus the NFS algorithm polls too infrequently to provide useful guarantees to the applications programmer, yet too frequently to yield minimal server and network loads.

2.3. The Apollo Domain System

The Apollo Domain system [Leach 83] uses an interesting interface to the file system: Domain system applications process files by mapping them into virtual memory. After mapping, normal memory reading and writing instructions are used to access the file data.

Objects are identified within the Domain system by 64 bit unique identifiers (UIDs). Associated with each object is a timestamp, indicating when the object was last modified. The basic operations performed by clients of the Domain file system are *read* and *write* operations, whose parameters include the UID and page number requested. *Read* requests return, along with the data, the timestamp associated with the file. If the timestamp indicates that the file has changed since it was first mapped into a

process, that process will take an exception on the first reference it makes that is satisfied from the new version. Similarly, a write request sends to the server, along with the data, the timestamp associated with the file before this write request was performed. If this timestamp does not equal the file's timestamp at the server, then an exception is raised in the latest writer's process. The server returns the new timestamp after every write, enabling the client to track the timestamp of files it is writing.

The Apollo Domain system makes extensive use of caching in order to achieve good performance. Apollo has taken a novel approach to cache consistency: the client program is responsible for purging stale data from local caches. To read a file, the application first requests the current timestamp of the file from the server, and then purges all files pages with older timestamps. This guarantees that the application will read the latest data, and will not take any synchronization exceptions, as long as a new writer does not enter the picture while this process is reading the file. Furthermore, this architecture allows a program to use old cached data, if the application considers old information satisfactory.

Requests to directories are handled somewhat specially in the Apollo Domain system, due to the large number of requests to search directories. Normally, one would obtain a lock on a directory, read the directory and finally release the directory, with each of these three operations resulting in a message being sent to the host storing the data. However, since this operational triplet is so common, the Domain system has an additional call that performs the three operations with one network message. This has no effect on the synchronization guarantees made by their system, except, of course, that it enables them to use their locking architecture for directories without paying an unacceptable cost in additional message traffic.

2.4. AT&T Remote File Sharing

The AT&T Remote File Sharing (RFS) system [Rifkin 86] is the prototypical remote system call implementation of a distributed file system. When, in the process of handling a file system system call, the Unix operating system encounters a file system partition exported by a different machine, the entire system call is encapsulated in a message that is transmitted to the partition's real home machine, where it is executed on behalf of the requesting machine.

RFS uses AT&T's STREAMS [Olander 86] network interface, the System V reliable transport networking layer, to provide its communications facilities.

The client/server interface used in RFS is actually the Unix system call interface. For each system call that can encounter a remotely-mounted file system, RFS defines a message that describes the request. When a particular system call encounters a mount point to a remote file system, the entire system call's parameters are encapsulated in this message, and the message is sent to the site actually containing the file. The operation is performed at the remote site, and the appropriate data is returned to the client. In most cases, after the reply arrives, the client has simply to return the error code and any results to the requesting user process.

In 1986, the AT&T RFS system did not do any caching of directory or file data at the various client machines, since remote requests were not handled by the client. Cache consistency was thus not an issue for this version of RFS.

However, more recent versions of RFS [Bach 87] allow a client to satisfy read requests from a local buffer under certain circumstances. Specifically, when one client writes a file that other clients are reading, the readers are notified that they must purge their caches of the appropriate file's blocks. This mechanism operates only while a client has a file open; after closing a file, a client may not re-use data in its buffers without first checking with the server to ensure the file's version has not changed.

This mechanism is quite similar to that employed in the Andrew file system. However, the Andrew file system essentially keeps track of clients' caches across opens and closes, greatly reducing the number of transactions that involve the file servers. The Andrew file system's cache validation mechanism is described more fully below.

2.5. LOCUS

The LOCUS Distributed Architecture [Popek] is one of the most ambitious of the file systems discussed here. Before discussing LOCUS in detail, we must introduce some terminology. Three separate classes of sites are relevant to a file access. The file's **storage sites** (or SS) store copies of the file's data. The file's **using sites** (or US) are those accessing the data at any one time. Finally, the file's **current synchronization site** (or CSS) is a single site responsible for performing various synchronization tasks that must be executed at a single site.

When a file is opened, a LOCUS process sends a message to the file's CSS, in order to perform various synchronization housekeeping chores. This interaction with the CSS results in the US's getting a version number for the file, which it uses in later read and write accesses in order to ensure that the SS it is communicating with has the sufficiently up-to-date information. This is true for directories being searched during pathname evaluation, as well as normal data files. Directory operations are, however, treated specially for operations originating at a SS for the file. In this case, communication with the CSS is skipped, and the process simply accesses the local copy of the data. This algorithm does not lead to the use of out-of-date data because the CSS propagates directory changes to all SS and US machines at the time the directory modification occurs.

Once a file has been opened, LOCUS two types of **tokens** to synchronize operations affecting the open file. Once such token, called a **file offset token**, synchronizes access to file descriptors shared among several processors, while the other token, called a **file data token**, synchronizes accesses to shared inodes. The details two token synchronization algorithms are differ slightly, but essentially work by granting a token that, while possessed, allows the grantee to perform an operation of a particular type, for example, changing an inode. Before a new token granting a particular right can be given out, tokens granting conflicting rights must be rescinded. The essential difference between the file descriptor and the inode token granting algorithms is that file descriptor tokens always grant exclusive access to the file descriptor, while inode tokens may grant either shared (for reading) or exclusive (for writing) access to an inode.

LOCUS does provide some optional support for atomic operations. In particular, one may open a file in a transaction mode, where none of the changes made to a file are visible until the close system call commits them. LOCUS, like the Cambridge File System, uses an algorithm based upon shadow pages in order to implement these transactions efficiently.

2.6. Summary

Most of the systems described above (RFS, LOCUS, and the Cambridge File Server) provide strong synchronization guarntees concerning their use of stale data. In particular, all provide essentially the same guarantees as Unix: as soon as any file system operation completes, the results of the operation are available to any other workstation in the network environment.

On the other hand, all three of these systems communicate with the synchronization site for a file on every file open. This is a potential system bottleneck.

The remaining two systems, Sun's NFS and the Apollo Domain system, do not, by default, provide very strong file system consistency guarantees. In the normal case, a program may, on one machine, make various changes to a file or directory, and programs on other machines will not see these changes until some time later. These systems do, however, make more effective use of caches on the client machines, since the information in these caches may be used without having to explicitly query the server as to the data's validity.

The Andrew file system, described in the next two sections, provides strong consistency guarantees along with the caching efficiencies of the NFS and Domain systems. These sections describe how we did this, and why we bothered.

3. The Goals of the Andrew file system

The Andrew file system was designed to provide the appearance a single, unified file system to a collection of approximately 5000 client workstations connected via a local area network. In the briefest terms, the Andrew file system must be fast and inexpensive. With so many workstations connected to

the file system, the file system must not become a new bottleneck, now that adequate processor cycles are available.

Furthermore, the Andrew system is supposed to provide inexpensive computation resources for students at a University. The workstation cost is expected to be eventually less than a few thousand dollars, and the incremental cost of adding a client to the file system must be considerably less than the cost of the workstation itself. Our goal was that a single file server should be able to provide service to approximately 50 clients. We assume that a file server is a normal workstation augmented with approximately 1 gigabyte of disk storage.

Since the Andrew file system is intended primarily for use by Unix workstations, it must provide enough functionality to support the Unix file system. The Unix file system does not provide any support for stable storage except for a system call that guarantees a file has been completely written to the disk. Similarly, Unix does not support any sort of transactional facility. Thus the Andrew file system need not, and does not, provide any specialized transactional facilities.

On the other hand, the Andrew file system does provide useful consistency guarantees. Foremost in this regard is the guarantee that after the completion of an operation, the next operation performed anywhere in the network will see the updated file system state. For instance, after a rename system call completes on one machine, no client on the network is able to open the file under its old name, and all are able to open the file under its new name. Similarly, after a file being written is closed, any client on the network opening that file reads the new contents of the file, never the old.

In short, with one exception, discussed next, the Andrew file system guarantees that after a file system call completes, the resulting file system state is immediately visible everywhere in the network.

To understand this exception, one must understand how programs read or write files under Unix. A Unix process examining or changing the contents of a file first opens the file, specifying whether the file will be modified. Next, the process performs a sequence of read and write system calls (the latter only if the file was opened for writing). Finally the process closes the file.

On a standard Unix system, after the completion of any write system call, the data written by that call will be read by any process reading from the same file. This is true whether or not the reader opened the file before or after the writer performed the write system call. Thus two different users of the same file see changes made to the file appear at the read and write system call granularity.

In the Andrew file system, however, the new data written to a file is not actually stored back at the file server until the file is closed, and data cached on the workstation is only checked for currency at the time a file is opened. Thus, when sharing a file, users of one workstation will not see the data written at another workstation until the first open system call executed after the writer closes the file.

We chose these semantics for very pragmatic reasons. First, very few programs are disturbed by seeing changes at an open/close system call granularity rather than at the individual read/write system call granularity. Actually, few programs are even prepared for the possibility of a file's changing while they are processing it. In addition, our kernel modifications do not allow us to intercept read and write system calls to files located in the Andrew file system, a restriction imposed for efficiency reasons. Finally, when most programs finish processing a file, they exit, closing all of their open files. Thus Unix generally sees a close system call issued with respect to a file when a program is finished processing that file.

In actual practice, using the open/close granularity for consistency guarantees instead of the read/write granularity has not caused us any troubles.

Of course, the above discussion explains why our consistency semantics are strong enough; it does not demonstrate that they are necessary. There are several reasons we chose to make these consistency guarantees. First, many people in our environment use several workstations simultaneously, using a window manager to maintain several windows on multiple machines. These users can effectively move from one machine to another simply by moving the cursor from one window to another. For the system to be comprehensible, it is important that when a program in one window announces that it has done something, the results are immediately available to programs running in any of the other windows.

We considered requiring users to explicitly request the latest version of a particular file, rather than guaranteeing that the file system simply provides that version. We rejected this alternative, since it would burden naive users with a considerably more complex system model. Rather than thinking of a file as containing certain data, independent of the workstation from which it is viewed, the user would have to understand how data propagates in our system.

This can turn out to be a considerable intellectual burden. After using a remote processor to compile and install a new remote procedure call package, for example, a user should not be required to explicitly invalidate all of the object files, header files, and documentation that had just been installed.

Given these goals, the next section describes our file system design and how it achieves them.

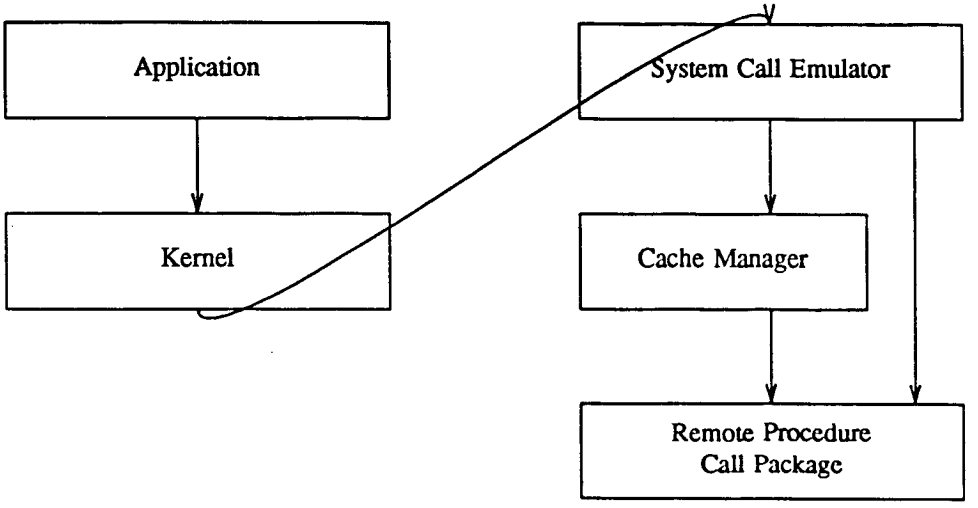

Figure 1. Venus Structure

4. The Architecture of the Andrew File System

The Andrew file system design embodies a small set of design principles.

Workstation caching
: Our primary means of minimizing file server load. Simply stated, workstations that already have most of their data cached on their local disks will not have to retrieve as much from the file servers.

Write-through cache
: A write-through cache minimizes the effects of workstation failure. Thus every time a file is modified on the workstation, a copy is sent back to the file server.

Minimize communication
: Network communications is moderately costly, and the less of it performed, the faster the resulting system will be. For operations that simply read data, the workstation should be able to handle the requests without contacting the file server at all, assuming the proper information is in the workstation's local cache.

Minimize server load
: When possible, perform processor-intensive operations on clients rather than on file servers; workstations have cycles to burn, while file servers quickly become bottlenecks.

Consistency
: We wanted our file system to provide the consistency model described above.

These five design priciples guided our entire design.

The Andrew file system consists of two types of machines, **clients** and **servers**. Clients run a program called **venus**, which essentially connects the workstation to the collection of file servers. Venus receives requests from the Unix kernel in the form of intercepted system calls, and translates them into operations on the workstation's local cache, as well as remote procedure calls to one or more file servers. Occasionally, *venus* receives RPC requests from a file server, notifying it that a callback, described below, has been broken.

4.1. Files and File Identifiers

Files are named at the vice/venus interface by 96 bit tokens called **fids**, for file identifier. Unlike some systems, such as Accent's Sesame file system [Jones 82], the Andrew system's fids do not name immutable objects: the contents of a file named by a single fid can have different contents at different times.

Files are comprised of a **data** part and a **status** part. The data part of a file consists of a stream of 8 bit bytes. The status part of a file contains various status information concerning the file, such as its length, the date the file was created, and the protection information associated with the file. One important piece of status information is called the **data version number**. A file's data version is incremented every time the file's data is changed, and is used by many of our consistency algorithms. The workstation caches the data part of a file and the status part of a file independently.

A file's fid is sufficient to retrieve the file's status or data parts from a file server at any time. Because files are not immutable, this file may contain different data at different times. However, the combination of a file's fid and its data version do uniquely specify a file's contents.

Files are stored on one or more file servers. Any file server can, when presented with a fid, return a list of those servers currently holding the corresponding file.

4.2. Directories and Vnodes

The Andrew file system can be view from two levels, the lower level being a flat name space, where each file is identfied by its *fid*, and the higher level being a hierarchical name space where files are identified by a pathname.

In designing the Andrew file system, we were concerned about the processor time required to translate pathnames to *fids*. We were especially concerned that file servers would not have enough processing resources available to handle extensive pathname-based computations. For this reason, we have tried to keep all knowledge of pathnames, as well as all requests that deal with pathnames, out of the file server interface. Instead, operations that deal with pathnames are handled by the workstation's *venus* process, with *venus* decomposing these operations into a series of operations at the *fid* level.

In more detail, the cache manager provides a useful abstraction: fast access to the data and/or status parts of files named by *fids*. The *venus* system call emulator is built on top of the cache manager, and invokes the cache manager when it needs to fetch the status or contents of a file. For instance, when translating a pathname into a *fid*, venus begins by fetching the latest version of the root directory (whose *fid* is determined at bootstrap time) into the cache. This directory is searched for the next component of the pathname, and this new file's *fid* is extracted. If this is the last component of the pathname, *venus* has now completely evaluated the pathname into a *fid* that can be passed either to the cache manager or, via the remote procedure call interface, to a file server. If this is not the last component of the pathname, *venus* iterates, searching this new directory for the next component of the pathname.

After the pathname has been reduced to a *fid* by the system call emulator, the emulator performs the requested operation on the *fid*. For example, if this is an *open* system call, *venus* will fetch the data part of the file into the cache, and arrange for further read and write operations to be directed to the file in the cache. If *venus* is emulating an *chmod* system call, it will issue a remote procedure call of the procedure *ViceStore*, with the *fid* of the target file, and the updated file status.

Cache consistency, in this architecture, amounts to ensuring that calls to the cache manager return up-to-date data from the cache. Rather than polling the file server on each call to the cache manager, or simply assuming that the data in the cache is up-to-date, the Andrew file system uses some state shared between the file servers and the client workstations to notify client workstations when their caches contain invalid information. This shared state is called the *callback database*, and is described in the next section.

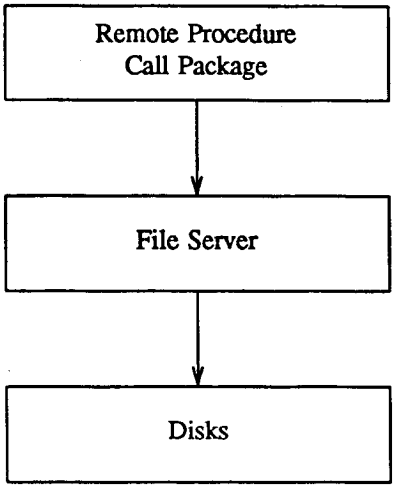

Figure 2. File Server Structure

5. Callbacks

The Andrew file system maintains cache consistency via **callbacks.** Simply, when an Andrew file system client fetches a file's data or status information from a file server, it may also receive a promise from the file server that the file server will notify the workstation before it changes the data or status information associated with the file. Such a promise is called a **callback.** The process of notifying the appropriate workstations when a file has changed, which must be performed before changing the contents or status information associated with a file, is called **breaking a callback.**

Note that when a workstation has a callback outstanding on a file in its cache, it can read the data or status associated with that file from its cache without any fear that the information is out-of-date. This considerably reduces the load that a workstation places on a file server, without weakening the cache consistency guarantees.

Note that while callbacks are essentially a simple concept, their implementation has some tricky aspects. These problems arise in two areas: synchronization of calls to the file server with callback breaking messages from the same file server, and communications failure while breaking a callback promise.

5.1. Callback Interface

This section describes how callback promises are made and broken via the client/server interface in the Andrew file system. In the Andrew file system, both the clients and the file servers act as remote procedure call servers. The file servers export the standard file server interface, while the clients export a simpler interface allowing file servers to break callbacks.

Only three calls in the file server interface involve callbacks in any way. The *ViceCreate* call creates a new file in a directory. It returns the *fid* of the newly-created file, and this *fid* is returned along with a callback promise. The *ViceFetch* call is used to fetch the status, or status and data parts of a file from the file server, and also implicitly returns a callback promise for the file involved. Finally, the *ViceDeleteCallBack* call notifies the file server that the client is no longer requires its callback

promise.

The callback interface exports several procedures for the file servers. The most important, of course, is *CallBack*, which directs the client to break the callback promise associated with a particular *fid*. There is also a procedure allowing the file server to break all callback promises associated with an entire set of files, for use when those files are moved from one file server to another.

Note that the only parameter passed from the file server to the client when a callback promise is broken is the *fid* of the associated file. We discovered that more information is required to avoid certain race conditions; this problem is discussed in more detail in the next section.

5.2. Callback Synchronization

The problems involved in handling callback-breaking messages in *venus* all occur when an outgoing to the file server returns a callback promise at the same time that the file server calls the client to break a callback promise associated with the same file. Due to the lack of precision in the client/server interface, *venus* can not tell if the callback-breaking message applies to the just-returned callback promise, or to a previous one, and thus *venus* does not know if it actually has a callback promise on the particular file.

There are several possible solutions to this problem.

One's first reaction when faced with synchronization problems of this nature is to consider locking the appropriate data structures. The structures involved in this case are the cache entries in the client, and one could lock cache entries when making requests that may return a callback promise, or when processing a callback-breaking message. The problem with this proposal is that it is very difficult to determine a valid locking hierarchy to avoid deadlock; we were unable to.

It is easy to understand the difficulty: most file system operations require locking the client's cache entry before making a call to the file server. Yet the file server is the only site with the knowledge of which callback promises must be broken during the operation, and thus which cache entries must be locked at other clients. The file server may discover, while processing a request, that the locking partial order has already been violated by the client initiating the request.

Unfortunately, in the situation where the file server discovers the locking hierarchy has been violated, the file server has no recourse but to break some client lock. And allowing for this possibility greatly increases the complexity of the client's code. Thus we ruled out the locking proposal as infeasible.

Another proposal was to add a version number to the callback database at the file server. All changes to the callback database are generated by the file server, thus the file server can order changes to the callback database, and mark all messages with an indication of which version of the callback database the messages refers. When a message arrives breaking a callback promise, while a remote procedure call is being processed that eventually returns a new callback promise, the two operations can be ordered at the workstation by simply comparing the callback database sequence numbers and processing them in the same order as the server.

An ad hoc solution is also possible, based upon the observation that a client may always discard a callback promise without affecting correctness. When a callback promise is returned by a call that overlapped the reception of a message breaking a callback promise for the same *fid*, venus can simply discard the callback promise.

While the callback sequence number solution is by far the best, we actually have implemented the ad hoc solution instead, since we discovered this problem while actually running the system, and fixed it with the fastest emergency patch we could design. Our estimate is that only a few callback promises are incorrectly discarded per week in an environment containing several hundred workstations, so we have never taken them time to implement the cleaner solution; rather we opted to extensively comment the ad hoc one.

5.3. Callback Failure

The callback algorithms described above depends upon the reliable delivery of messages to break callback promises. Temporary network outages can prevent the delivery of callback breaking messages; the ability to tolerate these outages is important.

One of the more virulent manifestations of this problem occurred with our printer spooling software, although many systems are vulnerable to these failures. In the Andrew file system, a workstation can run for an unbounded amount of time between calls to the fileserver, because the presence of a callback promise permits the client workstation to treat its locally cached information as correct. If a network failure occurs while delivering a callback-breaking message from a file server, then from the time the failure occurs until the time the client tries to contact the file server again, the workstation will not receive callback breaking messages. During this time, the client incorrectly believes it has the current version of some files. As soon as the client either successfully contacts the server, or fails to due to network troubles, *venus* will realize that it has not been receiving callback breaking messages.

To bound the time that a workstation can run with an out-of-date version of the callback database, and thus an out-of-date file, the *venus* process on every client workstation regularly probes each file server from which it has callback promises. Currently probes are sent every 10 minutes. When a probe from the workstation to the file server returns a callback database version number different from the version stored at the workstation, the client knows that a temporary network outage had lost some callback breaking messages, and that it must re-synchronize the callback database.

Bounding the duration of these error-induced inconsistencies is crucially important to certain programs. Our printer spooler, for example, is a very simple program that wakes up every 30 seconds and checks the contents of a directory for queue entries, printing any files referred to in those queue entries. When done printing a file, the spooler deletes the queue entry for the request.

Under normal operation, the printer spooler machine has a callback promise on the spooling directory, and the checks it performs every 30 seconds does not result in communication with the file server. When this callback promise is broken, however, the next time the spooler scans the directory, it fetches the new contents of the spooling directory. Were the callback breaking message lost due to a network outage, the printer spooler would never see a change to the spooling directory again. The addition of the periodic polling on behalf of the workstation means that for the printer spooler, a temporary network failure leads at most to a ten minute suspension of services after network operation has been restored.

6. Callbacks in Other Systems

This section describes how callbacks could be used to implement the stricter consistency guarantees made by RFS and other distributed file systems. As RFS provides the strongest guarantees of any file system discussed here, the remainder of this section discusses only RFS.

There are two areas where the Andrew file system does not provide consistency guarantees as strong as those provided by RFS. The first area is that of granularity: RFS guarantees that a read system call anywhere in the system always reads the latest written data, whether or not the file has been closed. The second area is locking: the Andrew file system does not support locking very efficiently--each lock call sends a message to the appropriate file server. In an environment where open calls implicitly set shared or exclusive locks, these extra messages would eliminate any advantages obtained from callbacks.

Our problem of checking the cache for stale data on with read/write system call granularity is more an artifact of our implementation than a fundamental design choice required by the callback architecture. For performance reasons, the Andrew file system does not intercept read and write system calls, and so can not perform cache validity checks upon their execution. In a kernel implementation, the Andrew file system could perform the same validity checks upon every *read* that it does upon every *open*, producing the same consistency guarantees at the finer *read/write* system call granularity.

The more interesting issue is the implemention of a correct locking protocol with minimal message traffic. In an environment where one can open a file, read some data and close that file, all without any network traffic, one would hope that setting and clearing the appropriate locks could also

be done without network traffic.

This can be done fairly simply by a form of "lazy-evaluation." When a process releases a lock, it need not actually communicate with the file server at the time. Instead, the workstation itself continues to hold the lock, assuming that a similar lock request may be forthcoming. If a process on the same workstation requests the lock, it can be immediately granted. If a process on another workstation requests an incompatible lock, the file server must first request the workstation give up the lock. As long as no process actually holds the lock at the time, the client can do this immediately; otherwise, the client must delay granting the file server's request until the user process is done.

In short, network traffic is required for obtaining a lock only when an incompatible lock was previously granted to a client on another workstation.

7. Conclusions

We have learned a great deal from our implementation of the Andrew file system, and many of these lessons can be applied in building any other distributed file system.

Of course, caching is critical for getting good performance from a distributed file system. Caching is also a very powerful tool for reducing the load a fixed number of workstations will place on a file server.

Moreover, powerful cache consistency guarantees are very useful for casual users of the system, as well as for programmers engaged in building applications. While these guarantees are not trivial to provide, they are well worth the effort. Even relatively naive users of our system use the Butler system [Nichols 87] to commands on otherwise idle machines. These users have come to expect that when their status monitoring program announces new mail has arrived, they will be able to read the new mail with the mail-reading program, without regard to the location of either of these programs. Our system maintenance procedures involve compiling programs on several machines (and machine types) concurrently; building the procedures to synchronize these processes was greatly simplified by our relatively strict file system consistency guarantees. Students have even written multi-user games using the Andrew file system to effectively provide shared memory (this application put a considerable load on the file server). And, of course, people writing and debugging distributed systems especially appreciate the convenience good consistency guarantees provide. In short, strong consistency guarantees are useful for many classes of users, from the most naive to the most sophisticated.

Finally, and most importantly, an efficient file system providing powerful consistency guarantees and which makes extensive use of caching can be built, using *callbacks* for synchronizing changes to files stored in workstations' caches. The use of *callbacks* in the Andrew file system handles between 97 and 99 percent of the requests that otherwise would have to contact the file server to ensure cache consistency. The Andrew file system is one example of such a file system currently in use by over 300 workstations on the Carnegie-Mellon University campus.

It is important to note that the callback architecture used in the Andrew file system is not essentially incompatible with the numerous distributed file systems surveyed in Section 2 of this paper. For example, the interface provided by NFS file servers is quite similar to that provided by Andrew file servers, and we believe that NFS could be extended to make similar use of a callback architecture. Remote open-style file systems, such as AT&T's RFS, could also use callbacks to reduce network traffic.

In summary, callbacks provide a tool applicable to a variety of distributed file systems, enabling a distributed file system to make strict guarantees as to the currency of cache information without requiring explicit communication with a common synchronization point at every reference. The use of callbacks in a distributed system allows it to provide both good performance and useful consistency guarantees.

8. References

1. M. J. Bach et al.
A Remote-File Cache for RFS
Usenix 1987 Summer Conference Proceedings: 273-279, June 1987 Usenix Association, P.O. Box 7, El

Cerrito, CA.

2. A. D. Birrell and R. M. Needham.
A Univeral File Server
IEEE Transactions on Software Engineering New York, Se-6(5), 450-453, Sept. 1980

3. A. D. Birrell and B. J. Nelson
Implementing Remote Procedure Calls
ACM Transactions on Computing Systems 2(1): 39-59, Feb. 1984

4. M. B. Jones, R. F. Rashid, M. Thompson
Sesame: The Spice File System
Internal Documentation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, 1982.

5. S. R. Kleiman
Vnodes: An Architecture for Multiple File System Types in Sun UNIX.
Usenix 1986 Summer Conference Proceedings: 238-247, June 1986 Usenix Association, P.O. Box 7, El Cerrito, CA.

6. H. F. Korth
Locking Primitives in a Database System
Journal of the Association for Computing Machinery 30(1): 55-80, Jan. 1983

7. Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson and Bernard L. Stumpf
The Architecture of an Integrated Local Network
IEEE Journal on Selected Areas in Communications Vol. SAC-1 (5): 842-857, Nov 83

8. James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal and F. Donelson Smith
Andrew: A distributed Personal Computing Environment
Communications of the ACM 29 (3): 185-201, Mar. 1986

9. R. M. Needham and A. J. Herbert
The Cambridge Distributed Computing System
Addison-Wesley Publishing Company, 1982

10. B. J. Nelson
Remote Procedure Call
Ph.D thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, May 1981

11. David A. Nichols
Using Idle Worktations in a Shared Computing Environment
to appear in the *Proceedings of the 11'th ACM Symposium on Operating Systems Principles* November 1987

12. D. J. Olander, et al.
A Framework for Networking in System V
USENIX Conference Proceedings: 38-46 Atlanta, Georgia, June 1986

13. Gerald J. Popek and Bruce J. Walker
The LOCUS Distributed System Architecture,
The MIT Press, Cambridge, Massachusetts

14. Andrew P. Rifkin, Michael P. Forbes, Richard L. Hamilton, Michael Sabrio, Suryakanta Shah, Kang Yueh
RFS Architectural Overview
Usenix 1986 Summer Conference Proceedings: 248-259, June 1986 Usenix Association, P.O. Box 7, El Cerrito, CA.

15. D. M. Ritchie and K. Thompson
The Unix time-sharing system
Bell Systems Technical Journal 57(6): Jul.-Aug. 1978

16. R. Rodriguez, M. Koehler, R. Hyde
The Generic File System
Usenix 1986 Summer Conference Proceedings: 260-269, June 1986 Usenix Association, P.O. Box 7, El Cerrito, CA.

17. M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector and Michael J. West
The ITC Distributed File System: Principles and Design
Proceedings of the Tenth ACM Symposium on Distributed Systems Principles 19(5): 35-50, Dec. 1985