

The Ring File System

John R Lane

September 6, 2000

Abstract

Most medium to large organizations are now utilizing the economies of scale created by the booming home personal computer (PC) market to fill desktop workstation spaces for their employees. By in large, however, these PCs come, as standard equipment, accoutred with enormously large hard disks. Most of these organizations, are utilizing some form of centralized distributed file system for file sharing, ease of setup, backup and maintenance. Given that there is no effective method in place for utilizing these local disks in a distributed fashion, this means the local (workstation) disk space is largely wasted – being delegated by default to the individual user’s MP3 whims and web-browsing wishes. This paper seeks to present one possible solution to this problem.

Contents

1	Introduction	1
2	Specification	2
3	System Specifics	3
3.1	Overview	3
3.2	File Transfer	3
3.3	Token Passing	4

3.3.1	Failures	4
3.3.2	Latency	5
3.4	The File System Map	5
3.5	Subsystem Interaction	5
3.5.1	Local writes	5
3.5.2	Consistency	6
3.5.3	Callbacks	6
3.5.4	Locking	6
3.5.5	Redundancy	6
3.5.6	Token Handling	6
3.5.7	Token Receipt	7
3.5.8	Ring Faults	7
3.5.9	Token Loss	7
3.5.10	Quick Token Handling	7
3.5.11	Quotas	8
3.5.12	Backups	8
4	Comparison	8
5	Conclusion	9

1 Introduction

This project began largely out of pure interest in novel distributed file system concepts and implementations. At first, I investigated more well-known “DFS’s” such as SMB, NFS, and even Web-NFS, in what was to be an attempt to compare and contrast them with newer DFS implementations such as AFS, and Coda. However,

the deeper I investigated the newer DFS implementations and, moreover recent papers on the topic, the more I desired to apply what I know of these things to create a new variation on a specific problem for which there is (to me at least) no known workable solution and which has not been thoroughly investigated by academia.

Necessity being the mother of invention, my problem is related to the fact that we, as a department, have a need for more undergraduate user disk space. In the past year, as part of a departmental lab upgrade, we have received from Sun Microsystems, 32 Ultra 10 workstations with 8.4 gigabyte disks. Only 1 gigabyte is used for the OS, and the rest (a grand total of $7.4 * 32 = 236.8$ gigabytes) is completely wasted. My goal was a system to utilize that nearly 237 gigabytes; one which provided a reasonably quick, and almost totally transparent interface to users, nearly all the administrative features we system administrators need, and built-in redundancy, since, after all, these workstations exist for the use of undergraduate instruction and cannot be depended upon for reliable service.

2 Specification

Let us begin a more formal investigation of the requirements of such a system by stating the basic underlying system assumptions we will be working from for the rest of the paper.

- There exists a reasonably fast network of workstations (NOW)
- Each workstation has much unutilized disk space
- There is a need for redundancy

- There is a need for user file system quotas
- File locking is desirable
- Session (“last close”) file semantics are acceptable
- File system latency should be negligible under normal use
- Files within the file system should be secure

The first two requirements are hardware-related, and should be increasingly easy to meet given the growth of the Internet and the growing size of PC hard disks. Redundancy is a given because of the nature of the system, but need not always be a necessity; a file system solely for temporary files, for example, might not truly require any redundancy. User quotas are a necessity for disk usage in most academic environments and can even be useful in maintaining system integrity even in systems where quotas are not strictly necessary.

File locking is something that can be worked around and is not an essential part of a complete DFS implementation as applications are free to implement their own effective locking schemes. However, it is useful, and as we will see, the mutual exclusion that it necessitates comes for free as a direct result of another part of our system, so we include it.

Session semantics allow aggressive caching of data without requiring a server to maintain state thereby allowing much better write performance. They may arguably be less intuitive and more difficult to program for than more traditional Unix (“last write”) semantics, but the performance benefits they allow outweigh these considerations and the fact that they have been used with success in other systems (Coda, AFS,

etc.) suggests that session semantics are not inhibitably clumsy.

We would also like the file system to behave as if it were local, so latency in accessing files on machines across the network needs to be addressed along with schemes to make browsing of the file system quick as well. Security of files against unauthorized access is both the responsibility of the protocols outlined here and of the underlying file system. We will assume a secure underlying file system, compatible with minimum set of our goals throughout this paper.

The protocols we shall present here rely upon a sound foundation and are thought to be secure, but peer review is certainly necessary. We aim for this file system to be as usable and maintainable as existing DFS solutions such as AFS or NFS. Finally, some of the mechanisms presented here for such subsystems as ring failure handling and version handling on fail-over are still the topic of much thought and research.

3 System Specifics

3.1 Overview

Before explaining any design decisions, or performing any comparisons with other systems, let us begin with a brief overview of the system. Files are stored on a dedicated partition of the hard disks of participating workstations across an interconnected network. Files are located via the use of a file system map which, among other things, maps file names to “server” locations. A copy of this map is held locally on each machine and is updated via a token passing mechanism. When an application attempts to open a file on a client, the client first looks the file up in its map, contacts the server (or servers in the case of a server failure in a replicated environment)

to which the file belongs, transfers the file locally, and from here the file is handled as if it were local (until a `close()` is performed on it). When an application attempts to close a file, if the application has not modified it, the client either simply discards the file, or if the file was modified (written to), writes a new version identification number for the file to the token and passes it on. Clients receiving the token along the way will make note of the newer version in their file system map and possibly update their version if they find it advantageous to do so. We now move on to a discussion of the mechanics of this system by laying out the subsystems that make it possible: file transfer, token passing, and the file system map.

3.2 File Transfer

File transfer is a very simple abstraction that attempts to assure that transferred files are being sent to authorized entities, and that incompletely transferred files do not cause file corruption. Usage of the abstraction is made possible via a client file transfer method.

Discussion of a secure implementation must be done with respect to some existing system. Although it is clear that a “server” could accept or reject any file request based on some negotiable client-provided parameter (some method for authentication), we will discuss a possible implementation on a system familiar to the author: Unix with secure RPC.

In keeping with the symmetry of the system with respect to each host being essentially identical to any other, a “flat” security model is utilized here – all clients must trust one another and rely upon each other for security. As we will later see, this decision does not really make our system less secure than many other production

systems.

The basic idea in this implementation is that secure RPC handles the authentication of machines and users. From there, an authorized user on an authenticated machine is allowed to trigger the transfer of a file locally, at which point the existent security permissions for the file (from the file system map) are placed on the file (in the underlying file system's native format).

Many other systems including, most famously, NFS rely upon the trustability of its client machines. In this sense RFS is no different. However, this is not to say that a more stringent security system could not be effected on top of the guarantees we provide. We would like for RFS to be as simple and thereby as extensible as possible. A system in which RFS is simply used merely as a means for the maintenance of a file system is easily envisioned: with one or more clients exporting the file system to users with a very different set of security semantics in place.

Protecting against file corruption from incomplete file transfer is not particularly difficult particularly given a reliable datagram service such as TCP, though more complex systems utilizing tougher checksumming could easily be implemented if there was some demonstrable gain to be had from doing so.

3.3 Token Passing

Token passing is the subsystem that allows for mutual exclusion where it is necessary within the system. As any savvy reader might well note, it does not come without its own problems. Recovery from token loss, and dealing with faults in the token ring are not simple problems. They are, however, problems that have been well studied, and for which there exist reasonable solutions. Latency issues must also be considered

here – nodes waiting for tokens may well cause a noticeable slowdown if proper measures are not taken to prevent this.

Before delving into the problems caused by token passing, let us restrict their domain to our own file system and in the process help us to more easily resolve them. Tokens in our system generally consist of only one type of information: file system map updates. File system map updates allow clients to keep their local maps up-to-date with changes in the file system, allow them to see changes made to files they may have open or cached, and allow replicated servers to see file updates that may make it necessary for them to retrieve a newer file version or at least update their map. From the information contained within the token, all file map data is maintained, including i-node information (dates, permissions, ownership, etc.), disk usage, user quotas, the list of clients within the system, and all file system operations can be performed (file creations, deletions, etc.).

3.3.1 Failures

A well-reasoned reader will note the importance of tokens in this system and its susceptibility to their failure. As seen previously, this can occur in two instances: token loss (a client failure while a client holds the token), or chain downage (the failure of a client which was to receive the token). As mentioned, many algorithms exist to overcome these problems; brevity demands that we discuss a few of them shortly but not dwell upon them.

The question of what to do in one of the two failure instances above is really one of what fail-safe guarantees are to be provided. For example, token loss can be handled in one of two ways: token regeneration on another client, or

by waiting for the failed client's recovery and subsequent token resend. The former solution provides users greater uptime but cannot guarantee that their data will be consistent (at least in the short term). The story is much the same for chain downage: if the token cannot be sent to the next client in the ring, the sending client can either choose to wait until the receiver recovers or can attempt to send to a host further down the chain, skipping the faulted host or hosts until such time as they become available. Again, the former provides quicker response time, but at the cost of a lesser consistency guarantee.

3.3.2 Latency

Finally, latency becomes an issue with systems having many nodes. As we will later see, we are careful here to not create a situation in which clients must hold the token for more than the time required to modify it in order to perform any operation. Therefore, the size of the chain, and thus the number of participating machines, should be allowed to become relatively large.

3.4 The File System Map

The file system map is the heart of the system. It allows clients quick access to all file system browsing operations (eg. `cd`, `ls`, and the like work as fast or faster than for local file systems due to the fact that all i-node information is held locally on the client), and allows clients to quickly recover from "server" malfunctions by seeking alternate servers on which to retrieve a file. The map follows the hierarchical layout of the file system itself – it is a graph in structure with the internal and root nodes being directories, and leaves being files (links are also stored as files). File nodes contain all information in the

underlying disk file systems' i-nodes, including such things as user and group ownership, permissions, dates, name, and size. In addition it contains a list of version numbers and an un-ordered list of entries of servers from which the file versions may be obtained.

3.5 Subsystem Interaction

Having explained the components of the system, let us now complete our specification by discussing their interactions and attempting to show how the system actually functions under normal and abnormal conditions.

Let us begin by specifying the "inputs" to the system; that is, the set of operations supported by our file system to an operating system. Files may be opened, read, written, closed, created, removed, locked, and unlocked. They may also have status requests performed upon them and have their attributes modified.

Direct client-client interaction is semantically quite simple – all other clients can be ignored and the problem becomes one between only two machines. Thus we will not cover it here, instead turning our attention to the trickier part of map maintenance and token passing; seeing what consistency guarantees can and should be made.

3.5.1 Local writes

A number of incarnations of file writes have been conceived in the design of this file system, but the most novel is by far the most promising. As mentioned before, file writes are only performed locally. Requested files are transferred locally, and modified locally. Upon file close, a new and unique version number for the file is released and propagated. Thus, the most harm a client crash

can do to files (assuming lower-level file system inconsistency detection) is to nullify file modifications made before the crash.

3.5.2 Consistency

It is not hard to see that this completely pushes write-consistency guarantees into the token's set of responsibilities because the token is the means of communicating a change to the rest of the system. The same is done with concurrency issues: the combination of versioning with local file writes assures that two writes can never occur to the same file in different places. There is the possibility of anachronistic inconsistency if, for example client *A* closes a file just before client *B* does, but client *B* gains token access first. This inconsistency, however, exists in various forms on other types of file systems and has been tolerated by DFS users for years.

3.5.3 Callbacks

Callbacks, or the informing of clients that a certain operation is to be performed on files they may hold open are clearly easy to implement: simply writing the desired action (as is specified by the protocol) to the token will inform all clients of the pending action. Given, however, that few applications can make use this feature, we believe it is still currently of limited use. A lack of callbacks, however, makes the unfortunate situation in which two users edit a file and then both simultaneously save their changes impossible to prevent or effectively handle without the use of file locking.

3.5.4 Locking

File locking is as simple as the rest of the system. Assuming the file is not already locked, a client

simply writes the lock (or unlock) of the file to the token, and all clients update their maps accordingly.

3.5.5 Redundancy

If a client should crash, the files stored on it are accessed through another client with the same requested version (usually the newest, but nothing requires it to be) and transferred from there. If no such client exists, the user must be informed as such.

Ensuring that redundancy is enforced is not trivial. The question becomes one of exactly where to store files. The most logical choice is to store them where they are needed, and the best way to achieve this is to wait for them to be requested and have them “stick” to the machines where they are used. This, however, makes the system vulnerable until the files requiring redundancy have been requested on the requisite number of clients.

To ensure greater redundancy, clients along the path could simply begin mirroring the file one by one until the requisite number of copies is achieved. Thereafter it would be wise for them to not update their versions on writes if a file is updated thereby allowing the file to find its best “home” locations. Other solutions certainly exist and redundancy is the subject of ongoing research.

3.5.6 Token Handling

As mentioned above, the problem of token handling in the presence of faults is one of making a choice between solid consistency guarantees and good performance. We believe that the most useful system is one that allows users to work most effectively regardless of faults, and

that most users are quite willing to accept a tradeoff of file inconsistency in a very rare set of cases to achieve increased usability and performance in more common ones. We therefore present a solution which chooses to ignore some consistency problems while giving users usability in the presence of faults. Obviously this is not the only solution; one can even envision multiple solutions being implemented in a given system, with the end-user (administrator) choosing the best for their site.

3.5.7 Token Receipt

First we must guarantee that the token is received properly and can recover from underlying client failure. We do this via a handshaking method when the token is passed between clients. The sending client waits for confirmation from the receiver that the token has been received and saved to stable storage. If it receives no confirmation of this, it retries enough times to presume the receiver has failed. From there the sender proceeds as if a ring fault has occurred.

3.5.8 Ring Faults

Ring faults are handled by the (stable) storage of token updates on the neighbor of the faulty client or network. It is this client's responsibility (in addition to its normal duties) to store updates (passed tokens) and check periodically to see if the neighboring host or network has resumed operation. Once a recovery occurs it is the client's job to forward the updates, en masse. This all-at-once token digest is tagged as such and treated specially: it is the responsibility of each client to read through and catch up – each client must look at each file update to check to

see if it holds a newer file update locally than the one it was given in the update digest. If it does, it should circulate a new version ID for its newer file. If a token comes during this time, it should enqueue it as an update and release the token, handling the new update after all digest processing is finished. Notably this method does rely upon clock synchronization to function correctly, and this is one of the tradeoffs discussed above. As explained previously, ring faults are still the focus of much current research.

3.5.9 Token Loss

We believe that token loss is a much less likely event than a ring fault and that the costs of overcoming it, which demand the addition of some manner of centralization to the system (at least for a time, for the purposes of determining who should regenerate the token) are not justified. We therefore propose that the system either wait for the failed client to return to service, at which point it will resend the token, or allow an administrator to intervene manually to recreate the token.

3.5.10 Quick Token Handling

Finally, in order to keep the system responsive, it is of utmost importance that tokens be handled as quickly as possible by each client. Clients must search through each token for relevant information, modify the token when they need to remove information that has already looped or update file system map information; much overhead can be incurred as a result. However, hashes and indexes into the passed data can be used to minimize this overhead. We do not anticipate, however, large amounts of data being passed within each token, and believe, there-

fore, that latency can be kept below a bounded threshold for a reasonable number of clients.

3.5.11 Quotas

Quotas are handled via the maintenance (as part of the file system map) of the number of bytes each user is using. Each time a file is modified, the client checks the modified file against the older version and adjusts and/or enforces the appropriate user's quota (the number of bytes in use versus the number of bytes the user may have in use) accordingly. Group quotas are handled in much the same fashion but with a total being made for each group for each member.

3.5.12 Backups

Backups are accomplished via the file versioning mechanism. The file map contains all version numbers up to an administrator-specified bound (beyond which files are deleted completely from the system). If a restoration to a former version is desired, doing so is as simple as transferring the file locally and circulating a new version ID for it.

4 Comparison

A full comparison with any of the myriad of existing DFS proposals and implementations is, for reasons of brevity, out of the question, so we, here choose to do limited comparisons with two existing systems: NFS and Coda. NFS is chosen not for any similarity in functionality of operation, but merely for its popularity. Given its extraordinarily widespread use, if a DFS system meets or exceeds NFS's functionality and ease of use, then it can be considered a reasonable candidate for its replacement (or at least its

usage) in at many environments. Coda is chosen because of its successor status to the largely popular advanced distributed file system AFS. By 'limited comparison,' we mean comparison to each of the two file systems observing a smaller set of measures: consistency guarantees, security, cross-platform usability, redundancy, locking, quotas, write semantics, and caching.

Coda provides a very good security model against both internal and external attacks; it does not require that any client machine be trusted and relies on the use of Kerberos tickets for user and client authentication. Security while built into our system, is left largely unspecified; for the most part because it is so platform dependent, relying, as it does, upon the underlying file system to provide its service. It is similar to NFS in this regard, and though NFS's security model has been rightfully challenged by many, it stands because of its ease of implementation and because the assumptions that it makes with regard to trusted clients are not always so difficult to guarantee for most of its users. Further comparison given the lack of specification is futile.

Cross-platform usability is one of the strong points of this system. Its simplicity, independence from its underlying file system, and the fact that it makes only basic assumptions regarding file structure (hierarchy, ownership, and permissions) should allow it to be implemented on a wide variety of platforms with relative ease. This is quite a contrast with Coda whose port from Unix systems to Windows NT, for example, has been an ongoing concern for a number of years. NFS is obviously the king of cross-platform usability, having been implemented on nearly every platform under the sun. Given that our system makes similarly few assumptions about the underlying system, file system layout and permissions, it should be equally well-served by them.

Attribute	Ours	NFS	Coda
Redundancy	R/W	R/O	R/W
Locking	✓	✓	✓
Quotas	✓	✓	✓
Semantics	Session	Unix	Session
Caching	File	Byte	File

Figure 1: DFS Comparison Matrix

The balance of the comparisons need rely little on prose and can be summed much more readably by the matrix shown in figure one. Attribute assignments reflect the most commonly used configurations.

5 Conclusion

We have shown the usefulness for file service via a networked set of workstations with largely underutilized disks, and presented a framework in which such service might be provided. This framework provides most modern file system features including low network latency, redundancy, file locking, and user quotas, while providing a feasible manner in which security may be implemented. Though security, token passing and caching schemes are left, to some degree, as implementation details, complete subsystems and semantics for their interaction have been specified and argued as correct with regard to the guarantees on consistency the file system makes. An implementation would certainly allow for more fine-tuning and experimentation and given the relative simplicity of the design, seems viable.

References

- [1] J. Bramme, “The Coda Distributed File System,” <http://www.coda.cs.cmu.edu/ljpaper/lj.html>
- [2] J. Bramme, “Coda Authentication and Protection,” <http://coda.cs.cmu.edu/doc/ps/sec.ps.gz>
- [3] Coda Maintainers, “Coda User and System Administrators Manual: System Administration: Volumes,” <http://www.coda.cs.cmu.edu/doc/html/manual-10.html>
- [4] T. Anderson, et. al., “xFS: A Wide Area Mass Storage File System,” University of California at Berkeley Computer Science Department, <http://sunsite.berkeley.edu/Dienst/Repository/2.0/Body/ncstrl.ucb/CSD-93-783/postscript>
- [5] T. Anderson, et. al., “Serverless Network File Systems,” University of California at Berkeley Computer Science Department, <http://sunsite.berkeley.edu/Dienst/Repository/2.0/Body/ncstrl.ucb/CSD-98-983/postscript>
- [6] B. Grönvall, A. Westerlund, S. Pink, “The Design of a Multicast-based Distributed File System,” Swedish Institute of Computer Science and Luleå University of Technology, <http://www.pdos.lcs.mit.edu/~jj/osdi/gronvall.ps.gz>