xFS: A Wide Area Mass Storage File System

Randolph Y. Wang and Thomas E. Anderson
{rywang,tea}@cs.berkeley.edu

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

The current generation of file systems are inadequate in facing the new technological challenges of wide area networks and massive storage. xFS is a prototype file system we are developing to explore the issues brought about by these technological advances. xFS adapts many of the techniques used in the field of high performance multiprocessor design. It organizes hosts into a hierarchical structure so locality within clusters of workstations can be better exploited. By using an invalidation-based write back cache coherence protocol, xFS minimizes network usage. It exploits the file system naming structure to reduce cache coherence state. xFS also integrates different storage technologies in a uniform manner. Due to its intelligent use of local hosts and local storage, we expect xFS to achieve better performance and availability than current generation network file systems run in the wide area.

1 Introduction

Recent advances in robot technology and tertiary media have resulted in the emergence of large capacity and cost effective automated tertiary storage devices. When placed on high performance wide area networks, such devices have dramatically increased the amount of digital information accessible to a large number of geographically distributed users. The amount of *near-line* storage and degree of cooperation made possible by these hardware advances are unprecedented.

The presence of wide area networks and tertiary storage, however, has brought new challenges to the design of file systems:

- *Bandwidth and latency*: At least in the short run, bandwidth over a WAN is much more expensive than over a LAN. Furthermore, latency over the wide area, a parameter restricted by the speed of light, will remain high.
- *Scalability*: The central file server model breaks down when there can be:
  - thousands of clients,
  - terabytes of total client caches for the server(s) to keep track of,
  - billions of files, and
  - petabytes of total storage.
- *Availability*: As file systems are made more scalable, allowing larger groups of clients and servers to work together, it becomes more likely at any given time that *some* clients and servers will be unable to communicate.

Existing distributed file systems were originally designed for local area networks and disks as the bottom layer of the storage hierarchy. They are inadequate in facing the challenges of wide area networks and massive storage. Many suffer the following shortcomings:

- *Poor file system protocol*: Frequently used techniques, such as broadcasts and write through protocols, while acceptable on a LAN, are no longer appropriate on a WAN because of their inefficient use of bandwidth.
- *Dependency on centralized servers*: The availability and performance of many existing file systems depend crucially on the health of a few centralized servers.
- *Data structures that do not scale*: For example, if a server has to keep cache coherence state for every piece of data stored on the clients, then both the amount of data and the number of clients in the system will have to be quite limited.
- *Lack of tertiary support*: Existing file systems do a poor job at hiding multiple layers of storage hierarchy. Some use manual migration and/or whole file migration. This is neither convenient nor efficient. Some offer ad hoc extensions to existing systems. This usually increases code complexity.

While others have addressed some aspects of the problem [2, 5, 9, 11, 13], there is yet no system that handles both WANs and mass storage. While we do not pretend that we have reached the perfect solution of providing fast, cheap, and reliable wide area access to massive amounts of storage, we present the xFS prototype as a design point from which some of the issues

This work was supported in part by the National Science Foundation (CDA-8722788), the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was supported by a National Science Foundation Young Investigator Award.

of mass storage in the wide area can be explored and future measurements can be taken.

Our goal in xFS is to provide the performance and availability of a local disk file system when sharing is minimum and storage requirement is small. We observe that many of the problems we face have also been considered by researchers studying distributed shared memory multiprocessors and there are many well-tested solutions that can be applied to our design. In particular, xFS has the following features:

- xFS organizes hosts into a more sophisticated hierarchical structure analogous to those found in some high performance multiprocessors such as DASH [10]. Requests are satisfied by a local *cluster* if possible to minimize remote communication.
- xFS uses an invalidation-based write back cache coherence protocol pioneered by research done on multiprocessor caches. Clients with local stable storage can store private data indefinitely, consuming less wide area bandwidth with lower latency than if all modifications were written through to the server. For data cached locally, clients can operate on them without server attention. The Coda [18] study illustrates that this should offer better availability in case of server or network failure.
- xFS exploits the file system naming structure to reduce the state needed to preserve cache coherence. Ownership of a directory and its descendents can be granted as a whole. By avoiding the maintenance of cache coherence information on a strictly per-file or per-block basis, xFS reduces storage requirements and improves performance.
- xFS integrates multiple levels of storage hierarchy. xFS employs a uniform log-structured management and fast lookup across all storage technologies.

xFS is currently in the middle of the implementation stage and we do not yet have performance numbers to report. The remainder of this position paper is organized as follows. Section 2 presents the hierarchical client-server structure of xFS. Section 3 describes the cache coherence protocol. Section 4 describes how xFS reduces the amount of cache coherence information maintained by hosts. Section 5 discusses the integration of multiple levels of storage. Section 6 discusses some additional issues including crash recovery. Section 7 concludes.

2 Hierarchical Client-server Organization

Andrew [6], NFS [17], and Sprite [20] all have a simple one-layer client-server hierarchy. While it is conceivable to port such a model onto a wide area network (figure 1), the result is likely to be unsatisfactory because it requires large amount of communication over the WAN. xFS uses clustering based on geographic proximity to exploit distinction between local and remote communication, utilize locality of data usage within clusters, and increase scalability of the system.

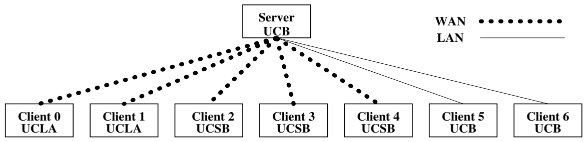

Figure 1: Simple client-server relationship.

The model seen in figure 1 has a number of problems. First of all, WANs have higher latency, lower bandwidth, and higher cost. Although the situation is expected to improve as the technology matures, we believe wide area communication is likely to remain more costly and offer less bandwidth than its LAN counterpart for some time. Even if the bandwidth bottleneck is solved, latency will continue to haunt a naive system. Even under the best possible circumstances, the round trip delay over 1000 km will be comparable to a disk seek. A simple client-server model as seen in figure 1 fails to recognize the distinction between wide area links and local links. The result is overuse of wide area communication, which in turn leads to high cost and poor performance.

Secondly, locality plays a more important role in a wide area where *clusters* of hosts that are geographically close to each other are likely to share information more frequently and have an easier time talking to each other. A file system that treats all clients as equals might make sense on a LAN where all clients are virtually peers that exhibit little distinction. Such an organization becomes less acceptable in a wide area where paying more attention to locality is likely to improve performance.

Thirdly, the organization of figure 1 has problems with scale. The server load and the amount of cache coherence state on the server increases with the number of clients. A straightforward deployment of many existing file systems in the wide area can be neither "wide" nor "massive".

In an attempt to solve these problems, xFS employs a hierarchy similar to those found in scalable shared memory multiprocessors (figure 2)[1]. To understand the motivation of such an organization, it is instructive to study the analogy between DASH [10] and xFS. The DASH system consists of a number of processor clusters. Communication among the clusters is provided by a mesh network. This is analogous to WAN links in xFS where bandwidth is limited and latency is high. Each DASH cluster consists of a small number of processors, analogous to clients on a LAN. Intra-cluster communication is provided by a bus. This is analogous to the LAN links in xFS where local communication is cheap and fast.

The analogy, however, does not stop here. Firstly, DASH recognizes the distinction between intra-cluster communication (bus based) and inter-cluster communication (point to point) to optimize performance. Sim-

[1] In principle, it is possible to extend the hierarchy further to form clusters of clusters.

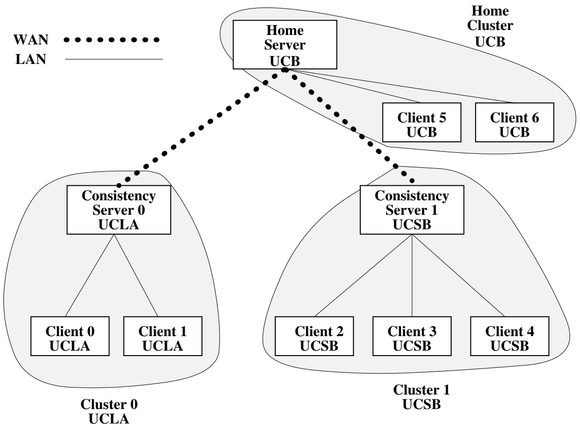

Figure 2: xFS hierarchy.

ilarly, xFS clients may choose to write through and/or broadcast more frequently on a LAN where communication is cheap and fast. When communicating across slower and more expensive WAN links, however, xFS hosts speak a write back protocol (section 3).

Secondly, the hierarchical organization does a better job at taking advantage of potential locality within clusters of workstations. When a DASH processor references data cached in a local cluster, no remote communication is incurred. Similarly, one experiment [2] has shown that roughly two thirds of the files read by one workstation are already cached by neighboring clients. A study done by Dahlin [4] shows clients within a cluster can effectively serve each others' cache misses. To exploit such locality, when an xFS client references data not present in its local cache, it first queries a *consistency server* on the cluster to see if any peer clients can supply the data. It is our hope that most of the local traffic can be contained within the cluster. Only occasionally will a consistency server have to contact the *home cluster* to handle cache misses.

The function of a consistency server is analogous to that of the directory mechanism in DASH which is responsible for locating data and enforcing cache coherence. Once the location of the desired data is discovered, data flows directly between the sender and the receiver. In a way, the separation of control messages[2] and data messages[3] in xFS is analogous to the approach taken in the Mass Storage Reference Model [3] where name services, file movement, and storage management all have distinct information flow paths.

A number of recent studies have explored the idea of hierarchical organizations [2, 5]. One experiment [5] employed an intermediate data cache server and its hit rate was found to be surprisingly low. In such a hierarchy of data caches, the higher levels are made of the same technology as the lower ones and the higher level caches contain little more than their lower level counterparts do. In effect, intermediate servers become "delay servers". An xFS consistency server, on the other hand, by not participating in data caching, avoids unnecessary delays.

Thirdly, the xFS hierarchy provides better scalability. Consistency servers shield the home servers from intra-cluster traffic. Furthermore, in an organization as shown in figure 1, the server must keep long lists of clients requiring consistency actions. In an xFS hierarchy, home servers will only have to track the consistency servers who act as agents for their local clusters.

We believe xFS has a hierarchy better suited for a wide area context. It recognizes the distinction of local versus remote communication. It exploits locality more aggressively. It distributes the burden of a centralized server to a number of consistency servers and peer clients. It is thus expected to offer better scalability.

3 Cache Coherence Protocol

Existing file systems use protocols that do not provide strong enough consistency semantics and/or generate excessive amount of network traffic. xFS speaks an invalidation-based write back protocol designed to minimize communication. A client with local stable storage can store private data indefinitely. The ability of a client to operate more independently without server intervention should also result in better availability.

NFS [17] writes through and offers little consistency guarantee for concurrent read write sharing. Andrew [6] provides a consistent view at file open time and writes back at close time. Sprite [14] enforces perfect consistency by disabling caching for write-shared files but still writes all dirty data to the server every 30 seconds for reliability reasons. Furthermore, none of these file systems does effective directory caching; consequently, even after optimizing write back policies, there are still a large number of name related operations left [19]. Such unnecessary use of network traffic might have posed little problem on a LAN but will become a performance bottleneck in a wide area.

We observe that the field of multiprocessor computer architecture has extensive literature in maintaining the consistency of cached replicas, while minimizing network usage by limiting communication to those cases when data is truly being shared. To reduce the number of bytes transferred over the wide area network, xFS uses a protocol based on multiprocessor style cache coherence [15]. An xFS host[4] requests *ownership* of a file or a directory from a higher level server. A client that possesses *read* ownership is allowed to cache data locally for reads[5]. A client that possesses *write* ownership is assured that it has the only valid copy of data. Such a copy is considered *private* and the host can cache

[2] This refers to messages related to requesting, granting, and revoking ownerships as discussed in section 3.

[3] This refers to messages that contain actual data.

[4] This refers to a client or a consistency server.

[5] Unix-like file access times are not kept.

and modify the copy indefinitely in its own stable storage without contacting the server. Ownership of data, once obtained by a consistency server, can be granted to lower level clients. A server, in order to enforce cache coherence, may refuse to grant ownership. It may also choose to revoke the ownership of data it has previously granted to other hosts. A client that has the write ownership is obliged to transfer its dirty data back only upon receipt of such a revoking call.

In existing *stateful* file systems [6, 14, 19], an `open` involves requesting ownership; a `close` involves relinquishing ownership; and a `callback` is associated with revoking ownership. Many systems do not distinguish the notion of `open` and that of requesting ownership, or the notion of `close` and that of relinquishing ownership. Even in a system that aggressively exploits caching such as Sprite [14], a client merely passes the user system calls to the server (figure 3a). If a user repeatedly opens and writes the same file (which is a likely event), a Sprite client dutifully transmits each open request and writes dirty data every 30 seconds.

xFS is different in two respects (figure 3b). Firstly, xFS explicitly separates the notion of `open` from that of requesting ownership. Ownership requests are sent only when necessary. Secondly, xFS protocol does not include anything that resembles a `close` system call. Hosts *never* voluntarily relinquish ownership. They give up ownership only in response to servers' revoking calls. If a user repeatedly writes the same file, only the initial open can result in a request for ownership and no further communication is required until another client requests the data. In this way, file and directory data is only transferred over the wide area network on a cache miss, a cache flush, or because of true sharing between multiple clients. In other words, xFS writes *on demand* as opposed to traditional file systems that write data through to guard against the possibility that they *might* be shared.

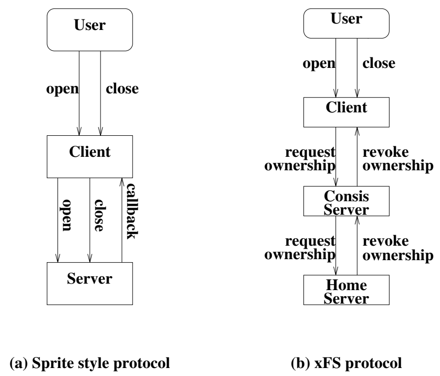

Figure 3: Separating notions of open/close/callback from those of requesting/relinquishing/revoking ownership.

Figure 4 shows the xFS file state transition in more detail. As an example, consider a file that is in the *read sharing* state. If a new host requests read ownership, it is granted the ownership and the file stays in the same state. If a host requests write ownership, the server revokes ownership from all the current readers. In the reply messages, the readers specify whether or not they still desire the read ownership. If none of the previous readers want to retain read ownership, the file goes into *write-1* state and the new writer is granted write ownership. If some readers are still actively reading the file, then the file goes into *write sharing* state and all hosts are denied ownership.

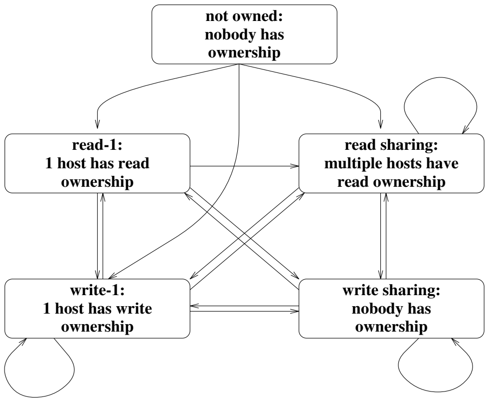

Figure 4: xFS file state transition.

Due to its better use of the local cache and less dependency on servers, xFS clients not only should observe lower latency and use less bandwidth, they should also achieve better availability. Existing file systems rely heavily on the servers for write throughs and name related operations. The demise of a single server or a failed network link usually renders the clients helpless. xFS hosts, on the other hand, after acquiring ownership of the proper data, can operate more independently. Furthermore, xFS hosts will have an easier time reintegrating after disconnection. A Coda [18] client, for example, is required to write the dirty data it has accumulated during disconnection back to the server during reintegration. In xFS, such writes are not necessary unless the dirty data is needed by others.

There are a number of problems with this cache coherence scheme. The first is the large amount of state information each host needs to keep. This is addressed in the next section. The second problem is the need for local stable storage that will keep dirty data indefinitely. This is addressed in section 5. The third is the need for policies to deal with network partitions and down hosts. This is discussed in section 6.

4 Reducing Cache Coherence State

Many existing stateful file systems maintain cache consistency in a way that is not designed to scale to handle massive amount of client cache. xFS reduces the cache coherence state by exploiting the hierarchical naming structure of the file system.

A Sprite or an Andrew server, for example, must keep track of *every* file cached by any client. The amount of server state grows with the total size of client caches. xFS lives on a wide area network and there are potentially large number of clients and large amount of callback information associated with each client. A way to deal with the explosion of state information needs to be found.

The hierarchical nature of the system as discussed in section 2 partially alleviates the problem. Home servers only need to keep track of the top level consistency servers. Bottom level consistency servers only need to tally the leaf clients that they serve. This reduces the length of client lists but does not reduce the number of files that might require consistency actions.

A second technique xFS uses to deal with state explosion is based on the observation that clusters of files tend to have the same ownership. There is little need to keep state for individual files when a representative for a large cluster can be used instead. Figure 5 illustrates the idea. Here we see user "foo" is working on machine "X". "X" has acquired read ownership for cluster "A" and write ownership for clusters "B" and "C". This has happened as a result of modification of her home directory and directory "baz" by user "foo". Similarly, machine "Y" has acquired read ownership for cluster "A" and write ownership for cluster "D". We see that ownership only needs to be associated with clusters of files.

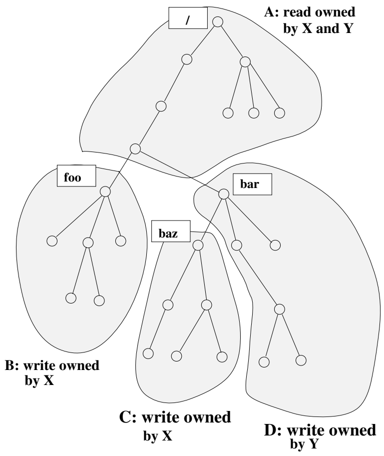

Figure 5: Hierarchical state information.

A third technique is based on the observation that a large number of files in the system are widely shared but rarely modified. For such files (an entire subtree that is exported for read only, for example), instead of remembering exactly which hosts have acquired read ownership, xFS merely remembers the fact that read ownership has been granted to *some* hosts. In the rare event that such files do get modified, xFS resorts to broadcasts to locate the readers. Such broadcasts, however, do not necessarily have to flood every single host due to the hierarchical nature of xFS. A home server communicates with all the consistency servers it serves and only the consistency servers that have acquired read ownership have to forward the revoking broadcast to their clients.

An xFS server maintains cache coherence state for clusters of files instead of individual files. For each cluster of files, a server only keeps a list of consistency servers instead of individual clients. And for files that are almost never modified, even this list can be done away with. By employing these techniques, we expect to substantially reduce storage requirements for cache coherence state.

5 Integration of Multi-level Storage

xFS is designed to utilize a multi-level hierarchy of stable storage, capable of indefinitely storing modified file and directory data. xFS integrates all storage devices in a simple and uniform manner:

- All storage devices are treated as caches.
- All storage devices are written in a log-structured manner.
- Data blocks are located by looking up virtual-to-physical *translation tables* maintained on fast devices.

In addition to deciding on appropriate migration policies that move data around, we need to address two immediate questions. The first of which is how to locate data as blocks migrate among different levels of storage. The second is how to lay out data on media in an efficient manner.

To solve the problem of locating data in multiple levels of storage, some existing systems extend the UFS data structures to incorporate tertiary devices. For example, a block address in Highlight [9] can belong to the disk farm, some tertiary store, or the disk cache for tertiary storage. Given an $\langle$`inode number`, `offset`$\rangle$ pair, locating data involves locating the inode and indexing the inode. If the block address is found to be on tertiary, a fetch is done to bring the missing data into the disk cache and the operation resumes. Such extensions usually complicate the code and force the new storage devices to inherit data structures that were originally designed for disks.

A solution to the second problem, namely that of laying out data on media efficiently, is provided by the *log-structured* file system (LFS) [16]. LFS appends newly written data to a *segmented log* and it is optimized for write performance. As old data is deleted, LFS reclaims disk space by recopying sparsely populated segments to the tail of the log and marks those segments as *clean*. LFS provides superior performance in environments where large main memory caches absorb most of the reads and consequently disk write speed becomes the limiting factor. Since many tertiary devices are append-only[6] and tertiary archival storage are

[6] Most media deliver best performance when they are used as append-only devices.

mostly write-only, log-structured layout is a natural way of managing such devices. Highlight [9], a descendant of LFS, bases its design on this observation. It employs a different cleaner process to migrate selected data blocks out to tertiary store, which is also written log-structured.

When designing the xFS approach to locating data, we draw analogies from solutions to memory management problems. A virtually addressed memory word can be anywhere from an on-chip cache to a secondary backing store. To find out where it is, we first have to translate a virtual address to a physical address. The xFS equivalent of a virtual address is a *block ID*:

file ID	block #

Such a virtual address can have its physical incarnation anywhere in a hierarchy of storage devices. Each device is treated as a cache. Each device employs a fast *translation table* that translates a block ID to a physical address on the device. A translation table entry is of the following form:

block ID	# of blocks	device address

As data migrate among different storage levels, we simply change the corresponding translation tables, a simpler and cleaner approach than extending the existing UFS data structures. This is similar to the approach taken by [7] where logical disk addresses are mapped to physical ones to allow a clean separation between file and disk management without sacrificing performance.

The memory management analogy, unfortunately, does not apply for data layout. Firstly, unlike processor caches which are usually direct mapped, we would like our xFS devices to be fully associative. Secondly, unlike processor caches and main memory, few of the storage technologies xFS manages are truly random access devices. For these two reasons, xFS writes its caches in a log-structured manner.

When an xFS device receives a read request, it queries its translation table to find the corresponding physical address. A successful search leads us directly to the device location where the data is found. If the search fails, the device declares a cache miss. When an xFS device receives a write request, it appends the data blocks to the end of the log and updates its translation table to reflect the new mapping. Invalidation of data blocks involves removing the corresponding mappings from its translation table. When it is time to clean, whether a block is live or not can be determined by comparing its identity against the corresponding translation table entry.

Ejecting from an xFS device is accomplished by a migration process. It chooses a number of victim blocks according to some policy. If the blocks are dirty, the migration agent retrieves the blocks from the source device and sends them to the destination device. Then it simply invalidates the blocks in the source device and the freed space will be reclaimed by the cleaner.

In principle, the translation table can always be "paged" to a slower device. For simplicity, we have opted to keep the translation table entirely on a faster device. For example, the translation table for a disk cache is kept entirely in memory[7] and the translation table for tertiary is kept entirely on disk. An obvious disadvantage of doing so is the storage requirement needed for the translation tables. For example, tens of megabytes of main memory might be needed to manage less than ten gigabytes of disk storage. This problem is partially alleviated by using a single translation table entry to point to a number of consecutively written blocks. Thus large sequentially written files will need little space in the translation table. We believe given the fast decrease in memory cost, the memory used for the translation table is money well spent in exchange for the resulting simplicity. Similarly, studies of some workloads suggest the use of less than ten gigabytes of disk space will suffice for "translating" around ten terabytes of tertiary storage and such an investment is probably worthwhile.

We plan to implement two interfaces for each kind of xFS device: a *storage interface* that allows the device to be used as a caching store, and a *translation interface* that allows the device to be used as a translation device. A disk, for example, has two interfaces that allow it to be used either as a storage device or as a translation table for tertiary. An appropriately chosen pair of slow and fast devices can be "glued" together to form a *storage bin*. Migration agents running on each machine chooses blocks to inject into or eject from the storage bins it oversees. Pieces of the same file can potentially spread across multiple levels of storage hierarchy over a wide area, reflecting the widely distributed and tertiary nature of xFS.

6 Other Issues

One question any distributed system has to answer is how it handles machine crashes and network partitions. An xFS client has to remember what ownerships it has acquired from which servers. An xFS server needs to remember what ownerships it has granted to which clients. Even after applying the state compression techniques described in section 4, the amount of information would still be too large to keep entirely in memory as Sprite [20] and Spritely NFS [19] do. Part of it needs to be written to stable storage. On the other hand, we cannot afford to log every ownership change to disk. Consequently, we need to recover the memory resident information lost in a crash.

The task is relatively simple for clients. Whenever a client commits data to stable storage, it conveniently logs the corresponding ownership. Any lost ownership in a crash will not have dirty data associated with it. In the worst case, the server might believe it has granted

[7] Upon reboot, the memory resident translation table is reconstructed by examining the identities of data blocks on disk.

ownership to certain clients which the clients have lost in crashes. Later when the server sends revoking calls to which the clients simply reply that they do not have it any more. A recovering server, however, has to recover the *exact* state it had before the crash. It appears that a server centric approach as done in [12] would work well. Under such a scheme, a recovering server directs its clients to help rebuild the lost server state. It has also been noted that such frequently updated and small amount of volatile state is a good candidate for inclusion in a stable memory that can survive machine crashes and the recovery cost can be kept to a minimum [1].

Another question we need to answer is how to deal with machines that do not respond to ownership revokes due to crashes or network partitions. There are three alternatives. We fail the operation that resulted in the revoke in the first place, hang the operation indefinitely until the revoke is answered, or let the operation proceed at the risk of allowing consistency conflicts which will have to be resolved when the offending hosts rejoin. We adopt the third approach taken by Coda [18] which trades quality for availability. Fortunately, the Coda study has shown that such conflicts are extremely rare events and we do not expect ownership revokes to be frequent in xFS.

xFS's approach of storing data locally and indefinitely also raises other concerns. Security is one. Andrew [6] treats servers as first class citizens and clients are deemed less trustworthy. In xFS, however, the line between clients and servers are blurred when clients are allowed to serve each other (section 2). In such an organization, the clusters can be treated as security domains where cluster members trust each other but foreign clusters that are not homes are deemed untrustworthy. A more paranoid approach would require the writer to compute a checksum. Kaliski [8] has devised an efficient algorithm which makes it computationally infeasible to find two messages with the same checksum or a message with a prespecified checksum. A reader, upon receipt of the data and the checksum, can run the same algorithm to verify that the data has not been altered.

Another difficulty is backup. When the storage is concentrated on several gigabytes of disks on a few servers, disaster recovery (as opposed to crash recovery) can be accomplished by rolling the whole world back to a tape dump. Such an approach fails to work when there are massive amounts of storage and/or the storage is so widely distributed that taking consistent snapshots is virtually impossible. One possible way of providing disaster recovery for xFS is to avoid a disaster in the first place by insisting on having two copies of everything at different sites at all times. An effective way of providing backup for mass storage in the wide area is a topic for future research.

7 Conclusion

Existing disk based local area file systems can no longer meet the demands of mass storage over a wide area. xFS uses a hierarchy that can better take advantage of the locality nature. It speaks a cache coherency protocol that minimizes the use of wide area communication. Consequently, it is expected to operate with lower latency and consume less wide area bandwidth. xFS minimizes cache coherence state information by exploiting the hierarchical nature of file system name space and hierarchical nature of the cluster based organization. xFS integrates multiple levels of storage in a uniform manner. Its ability to take advantage of local storage should deliver better economy, superior performance, and higher availability.

Acknowledgements

Discussions with Mike Dahlin and Dave Patterson have helped improve the xFS design, particularly that of the hierarchical client-server structure. We would also like to thank Soumen Chakrabarti, Mike Dahlin, John Hartman, Arvind Krishnamurthy, and Clifford Mather for their helpful comments.

References

[1] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the Unix Environment. *USENIX Association Summer 1992 Conference Proceedings*, pages 31-43, June 1992.

[2] M. Blaze and R. Alonso. Toward Massive Distributed Systems. *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 48-51, April 1992.

[3] S. Coleman and S. Miller. Mass Storage System Reference Model: Version 4. IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

[4] M. Dahlin. Private Communication. 1993.

[5] D. Muntz and P. Honeyman. Multi-Level Caching in Distributed File System. *USENIX Association Winter 1992 Conference Proceedings*, January 1992.

[6] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51-82, February 1988.

[7] W. de Jonge, M. F. Kaashoek, and W. Hsieh. The Logical Disk: A New Approach to Improving File

System Performance. *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993. To appear.

[8] B. Kaliski, Jr. The MD4 Message Digest Algorithm. *Workshop on the Theory and Application of Cryptographic Techniques Proceedings*, May 1990.

[9] J. Kohl and C. Staelin. HighLight: Using A Log-structured File System for Tertiary Storage Management. *USENIX Association Winter 1993 Conference Proceedings*, January 1993.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.

[11] F. McClain. DataTree and UniTree: Software for File and Storage Management. *Digest of Papers, Proceedings of 10th IEEE Symposium on Mass Storage Systems*, pages 126-128, May 1990.

[12] J. Mogul. A Recovery Protocol for Spritely NFS. *Proceedings of the USENIX Workshop on File Systems*, May 1992.

[13] J. Mott-Smith. The Jaquith Archive Server. UCB/CSD Report 92-701, University of California, Berkeley, September 1992.

[14] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134-154, February 1988.

[15] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348-354, June 1984.

[16] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *Operating Systems Review*, 25(5):1-15, October 1991.

[17] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. *Proceedings of the 1985 Summer USENIX Conference*, June 1985.

[18] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *ACM Transactions on Computer Systems*, 39(4):447-459, April 1990.

[19] V. Srinivasan and J. Mogul. Spritely NFS: Experience with Cache Consistency Protocols. *Proceedings of 12th Symposium on Operating Systems Principles*, pages 45-57, December 1989.

[20] B. Welch. The Sprite Distributed File System. PhD Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, March 1990.