Safe Caching in a Distributed File System for Network Attached Storage

Randal C. Burns, Robert M. Rees
Department of Computer Science
IBM Almaden Research Center
{randal,rees}@almaden.ibm.com

Darrell D. E. Long
Department of Computer Science
University of California, Santa Cruz
darrell@cs.ucsc.edu

Abstract

In a distributed file system built on network attached storage, client computers access data directly from shared storage, rather than submitting I/O requests through a server. Without a server marshaling access to data, if a computer fails or becomes isolated in a network partition while holding locks on cached data objects, those objects become inaccessible to other computers until a locking authority can guarantee that the lock holder will not again directly access these data. We describe a server that acts as the locking authority and implements a lease-based protocol for revoking access to data objects locked by an isolated or failed computer. When a lease expires, the server can be assured that the client no longer acts on locked data, and can safely redistribute locks to other clients. During normal operation, this protocol invokes no message overhead, and uses no memory and performs no computation at the locking authority.

1. Introduction

A distributed system provides an operating environment to its attached computers. To be useful, this environment generally has some semantic and performance guarantees. These often include message delivery semantics, message ordering, failure detection, and availability and consistency of distributed data views. For such a computing environment to be *safe*, the guarantees must hold when components of the system fail.

The *Storage Tank* distributed file system includes guarantees for the continuous availability and sequential consistency [19] of all data in the system, including data cached at clients. In this work, we present a protocol that ensures the safety of these guarantees in the presence of network partitions. The protocol is based upon the concept of using *leases* [11] to synchronize actions between a client and server after communication between them has failed. In our system, a lease is contract between a client and a server in which the server guarantees the correctness of the data in the client's cache for the specified period. Furthermore, the client promises to not operate on cached data when it does not hold a valid lease.

This protocol does not address all elements of safety. In particular, the protocol does not protect against failures in the storage subsystem, which can keep data from reaching disk, and it does not address how we guarantee the availability of data from servers. Other mechanisms are needed to address these failures.

1.1. A Storage Area Network File System

A brief digression into the file system architecture in which we implement safety helps to motivate our solution and its advantages.

In the Storage Tank project at IBM research, we are building a distributed file system on a storage area network. A SAN is a high speed network designed to allow multiple computers to have shared access to many storage devices. The goal of Storage Tank is to take advantage of SAN hardware to provide computers with the scalability and management benefits of a distributed file system, without the performance penalty expected from client/server file systems.

For our distributed file system on a SAN, clients access data directly over the storage area network. Most traditional client/server file systems [29, 20, 16, 7] store data on the server's private disks. Clients function ship all data requests to a server that performs I/O on their behalf. Unlike traditional file systems, Storage Tank clients perform I/O directly to shared storage devices on a SAN (Figure 1). This direct data access model is similar to the file system for Network Attached Secure Disks [10], using shared disks on an IP network, and the Global File System [23], for SAN attached storage devices. Clients communicate with Storage Tank servers over a separate network, called a control network, to obtain file metadata. In addition to serving file system metadata, the servers run distributed protocols for cache coherency, authentication, and the allocation of file data.

Unlike most file systems, metadata and data are stored separately [21, 6, 13]. Metadata, including the location of

0-7695-0574-0/2000 $10.00

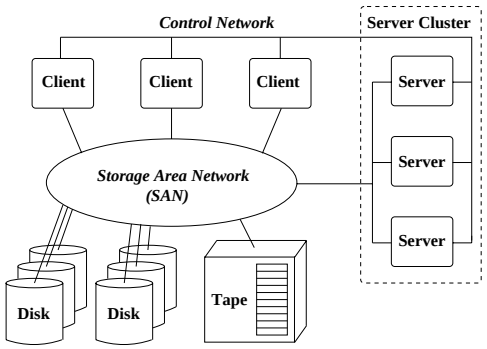

Figure 1. Schematic of the Storage Tank client/server distributed file system.

the blocks of each file on shared storage, are kept on high-performance storage at the server. The shared disks contain only the blocks of data for files, and do not contain metadata. In this way, the shared devices on the SAN can be optimized for data traffic, block transfer of data, and the server private storage can be optimized for metadata workload, frequent small reads and writes.

The SAN environment simplifies the distributed file system server by removing its data tasks, and radically changes the server's performance characteristics. For servers that perform data I/O, performance was measured by data rate (megabytes/second). Performance is occasionally limited by network bandwidth, but more often limited by the server's ability to read data from storage and ship it to clients on a network. In the SAN environment, a server's performance is more properly measured in transactions per second, analogous to a database server. Without data to read and write, the Storage Tank file server performs many more transactions than a traditional file server with equal processing power.

1.2. Failures in the SAN Environment

The Storage Tank server takes responsibility for maintaining data integrity and consistent views of data across all clients. This task is significantly complicated when components of the distributed system fail. One such problem arises when a computer that holds locks to access data loses contact with its server. Such a computer is unreachable and has either failed or is *isolated* from its server, separated by a partition of the control network.

The server cannot know the state of the locks and cache contents of an unreachable client. Either the computer has failed, and the locks will not be exercised again, or the control network between the server and the computer has partitioned. An isolated computer may be unaware that it is isolated and continue operating on locked data.

In many file systems, servers *steal* locks from unreachable clients to make data highly available [7, 16]. A locking authority steals a lock when it stops honoring a client's lock without notifying the client, and gives a conflicting lock on the same data object to another client in the system. Stealing locks is desirable when client computers become unreachable, because it makes locked data avaliable to other computers in the system. The alternative to stealing locks is to honor the locks of unreachable clients, which renders locked data unavailable indefinitely.

For file systems that read and write data through the server, a server may safely steal locks from an unreachable computer, regardless of whether the computer is failed or isolated. It is safe to steal locks from a failed computer, because that computer has lost volatile state and no longer functions as a client. For a computer partitioned from its server, isolation from the server is equivalent to isolation from the data store, because the server owns the storage. If the network partition merges, the client might again request service on a stolen lock. In this event, the server merely denies the request.

File systems for SANs, or other network attached storage environments [9], cannot safely steal locks from an isolated client. Unlike a failed computer, a computer in a network partition retains its lock state and may attempt to operate on file data after it becomes unreachable. Only the control network has failed, not the storage area network, and a computer in a network partition continues to read and write data to storage devices. If the locking authority were to steal the clients' locks, multiple clients, the old and new lock holders, could concurrently act on the same data without synchronization. The consistency and structural integrity of the file system would be compromised.

A currently accepted solution to this problem is to *fence* the isolated client before stealing its locks. To fence a computer, the server instructs the SAN-attached storage devices to no longer accept I/O requests from the isolated computer. Fencing may also be performed by instructing a switch to not route traffic for certain clients. The SAN devices must enforce this denial of access indefinitely. We will argue (Section 2.1) that fencing alone is not an adequate solution to this problem. As the only recovery mechanism, fencing violates the sequential consistency and cache coherency guarantees expected from a file system.

In addition to fencing, we propose a lease-based protocol that Storage Tank uses to make data highly available and safe in the presence of client and network failures. In our system, leases are held by clients and granted by servers. A held lease guarantees the correctness of the contents of a client's cache. Also, when not renewed, leases are used by servers to invalidate by time-out the cache contents of clients they cannot contact. This protocol allows an isolated computer to realize it is disconnected from the distributed system, and write its dirty data out to storage before its locks are stolen.

This protocol also has performance advantages when compared to other lease-based protocols [11, 27]. During periods of operation without failure, the protocol invokes no computational or message overhead, and requires no additional memory at the locking authority. Clients renew leases opportunistically through existing protocol messages, and servers are entirely passive, keeping no state about client leases until a communication error occurs.

2. A Two Network Problem

Due to the differences between the SAN environment and other distributed environments, Storage Tank addresses a different class of problems for maintaining data consistency in the presence of failure than traditional file systems.

Most previous work addressing safety in distributed systems assumes that network partitions form symmetrically [8, 5], *i.e.* if we define $V(A)$ to be all computers in the network view of computer A, then symmetric partitions imply that

$$A \in V(B) \text{ and } B \in V(A) \Leftrightarrow V(A) = V(B). \quad (1)$$

While reasonable for most distributed systems, this assumption does not hold in the SAN environment. Storage Tank uses two networks, a general purpose network for control operations, and a SAN for data traffic. Computers, clients and servers, can only communicate with each other on the control network. Storage devices only operate on the SAN. Anytime a partition, symmetric or otherwise, forms in either network, the two networks considered together partition asymmetrically. If the control network partitions between clients C_1 and C_2 (Figure 2), the disk is in each of the clients view, and the clients are in the disks view, but the views all differ: $D \in V(C_1)$ and $C_1 \in V(D)$ but $V(C_1) \neq V(D)$, because $C_2 \in V(D)$ and $C_2 \notin V(C_1)$. Similarly, if the SAN partitions, the two clients that contain each other in their network view have different views of the SAN. With two networks, a symmetric partition in one network can result in an asymmetric partition when views are considered across both networks.

Recent work has developed safe computing protocols that handle asymmetric partitions [22, 3], but this work requires all participants to run the protocol. In the SAN environment, the storage devices are often disk drives, and cannot maintain state and initiate messages as needed for safety protocols. Despite significant research support for distributed processing on active or intelligent disks with general purpose computation engines [17, 24, 1], these devices are unlikely to be commercially available in the near future. Disk drives on a SAN cannot execute non-storage code and consequently cannot maintain views and send data messages as required.

A simple example suffices to show how asymmetric partitions and the limited capabilities of disk drives result in data consistency problems when a network becomes partitioned (Figure 2). Assume a small Storage Tank installation consisting of a server, two clients and a single disk drive. Let one client (C_1) have an open instance with a data lock that permits it to read and write a file. At some point, the control network becomes partitioned. The server and the other client C_2 can no longer see C_1, but C_1 can still read and write data. If C_2 requests permission to write the same file, the server cannot successfully demand the lock, since it cannot communicate with C_1. Stealing C_1's lock and granting access to C_2 would result in multiple writers without synchronization. Caches would be incoherent and data integrity lost. The partition results in the server denying C_2 access to the file.

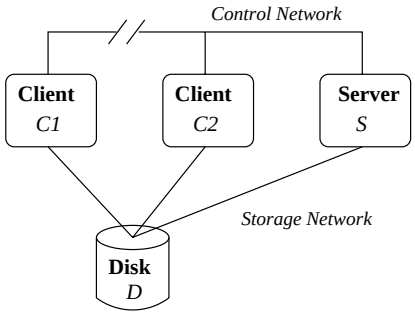

Figure 2. A two network storage system with a partition in the control network.

Without a suitable safety protocol, when a client enters a partition holding a locked file, no other client can access that file until the partitioned client reappears on the control network. Something as simple as a network partition can render major portions of a file system unavailable indefinitely.

2.1. The Inadequacy of Fencing

Adding the ability to fence clients, instructing a disk drive to deny service to particular initiators, does not solve all data integrity and cache coherency problems. Returning to our previous example (Figure 2), if the disk drive supported fencing, the server could instruct it to not allow C_1 to perform any I/O. Then, the server could grant C_2 write access to the file. While fencing does prohibit C_1 from performing concurrent conflicting writes, it leaves several opportunities for the violation of file system semantics.

Problems arise because the isolated client can have unwritten dirty data at the time that it is fenced. Most distributed file systems, including Storage Tank, allow clients to perform write-back caching, where clients write data to their local cache and the changed data are written back to persistent storage at some later time. In our example, after fencing C_1, dirty data on C_1 are stranded and never reach disk. If C_2 reads this data, it reads the old version from persistent storage. This is in violation of our consistency

guarantee – C_2 should properly read the most recently written version of data.

Also, the isolated client cannot determine immediately when it is in a network partition, and can potentially continue operations on a stale cache. The isolated client is out of touch with the server, and therefore does not receive cache invalidations. Also, C_1's locks have been stolen and the cache contents are not protected. Until C_1 attempts to perform I/O to the SAN, it is unaware that it is fenced. Local processes continue to read and write data out of the cache, and any of these data may have been modified on another client.

While fencing can ensure that clients in the server's and disk's network view have synchronized access to data, it does not provide the desired data consistency and cache coherency guarantees. Fencing fails both in that it prevents dirty cache contents from reaching persistent storage, and, it allows fenced clients to operate on stale cached data without detecting or reporting an error.

3. A Protocol Based on Leases

A safety protocol that protects the consistency of data and guarantees cache coherency must 1) protect the contents of a file by preventing isolated clients from writing data to disk after their locks are stolen, and 2) allows isolated clients to write the dirty contents of its cache to shared storage before its locks are stolen. Fencing fails to address the second point and clients behave incorrectly, giving stale data to local processes and losing written data. Storage Tank uses a protocol based on leases, similar to the leases defined in the **V** operating system developed at Stanford [11], that allows clients to determine that they have become isolated from the server. Upon determining their isolation, clients have the opportunity to write dirty data to disk before discarding their cache contents and ceding their locks. Leasing improves the semantics of failure and recovery when clients become isolated. For client failures or failures of the SAN, leasing offers no improvements over fencing.

Our lease-based safety protocol has performance advantages when compared to other leasing systems. These include optimizations that allow leases to be renewed opportunistically, eliminating message traffic during normal operation, and a design that allows for a "passive" server that participates in the protocol only when an error occurs.

Before developing the protocol, we state the network environment and assumptions required for the protocol to operate correctly. The goal of the protocol is to ensure data integrity and cache coherency among clients accessing network attached storage devices. To this end, our protocol addresses arbitrary partitions in the control network, including asymmetric partitions. Our protocol requires clocks at the clients and servers that are rate synchronized with a known error bound δ, *i.e.* an interval of length t when measured on one computers clock has length that falls within the interval $(t/(1+\delta), t(1+\delta))$ when measured on the clock of another machine. It does not require absolute or relative time synchronization, or Lamport clocks [18]. The protocol operates in a connection-less network environment, where messages are datagrams. However, many messages between the Storage Tank client and server are either acknowledged (ACK) or negatively acknowledged (NACK), and include version numbers for "at most once" delivery semantics.

A lease in this protocol defines a contract between a client and a server in which the server promises to respect the clients locks for a specified period. The server respects the contract even when clients are unreachable. A client must have a valid lease on all servers with which it holds locks, and cached data become invalid when a lease expires.

Servers use leases to time-out client locks. If a server attempts to send a message that requires an ACK from a client, and the client does not respond, the server assumes the client to be failed.

Clients that are isolated instead of failed have missed a message. They either have stale metadata or have not properly participated in a locking protocol. However, this does not yet result in a violation of file system semantics. Missing a metadata message is tolerable, because file systems only guarantee that metadata are weakly consistent[1]. Missing a lock protocol message is also acceptable, because locks and locked data are protected until the lease expires.

Having decided the client is failed, the server starts a timer that goes off at a time $\tau(1+\delta)$ later, where τ is the contracted lease period. The server knows that $\tau(1+\delta)$ represents a time of at least τ at the client. Once the server waits out this timer, it may steal the clients locks. The client is responsible for ensuring that all dirty data are written out by this time, and that the data and metadata it caches with respect to this server are invalid.

The key feature of the server's protocol is that it retains no state about client leases. During normal operation, the server merely grants locks and ignores leasing altogether. No lease-specific operations are performed and no server storage used. Only when a delivery error occurs does the server get involved by starting a lease timer. This passive design simplifies the implementation of the server and limits the performance impact of leasing.

3.1. Obtaining a Lease

For the server's decision to steal locks having waited time $\tau(1+\delta)$ to be correct, the client must be aware that its lease has expired before that occurs. Because clocks are rate synchronized by a factor of δ, a client whose lease starts before the server starts its timeout counter provides an ade-

[1] Modifications to metadata by one process are guaranteed to be reflected on the metadata view of another process eventually, but no instantaneous consistency guarantee exists.

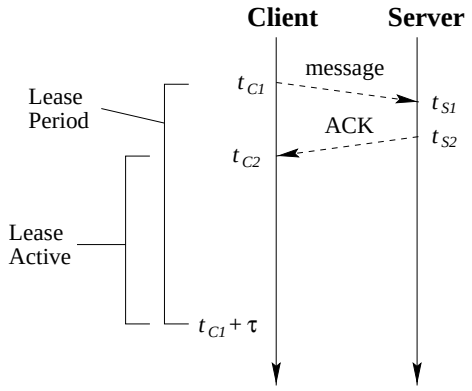

Figure 3. Client lease renewal.

quate guarantee. Since our system does not keep an absolute time, we use ordered events, a client to server message and the server ACK, to ensure that the server timer begins after the last lease that the client obtained.

A client implicitly obtains or renews a lease with a server on every message it initiates (Figure 3). At t_{C1}, the client sends a message to the server. The server receives this message at t_{S1} and acknowledges receipt at t_{S2}. Without synchronized clocks, the client and server have an absolute ordering on events $t_{C1} \leq t_{S2}$. Upon receiving the ACK at t_{C2}, the client obtains a lease valid for the period $[t_{C1}, t_{C1} + \tau)$. This lease's period starts from when the client initiates the message, not when it receives the ACK, because initiating the message is known to occur before the server's reply. While this lease is valid from $[t_{C1}, t_{C1} + \tau)$, from the client's perspective, it does not activate until t_{C2}, when the client receives an ACK, indicating it is in contact with a server. The client operates under this lease for the period $[t_{C2}, t_{C1} + \tau)$.

For correctness, we require the server not to ACK messages if it has already started a counter to expire client locks (see §3.3). This ensures that the server cannot steal locks from a client until after the client lease expires.

Clients are not granted leases when servers initiate communication. Because the client did not send an initial message, it has no discrete event that is known to precede the server message, and cannot safely choose a point in time at which the lease starts.

Theorem 3.1 *If a client and server have rate synchronized clocks by a factor of δ, the server cannot steal locks before the client lease expires.*

Proof. Referring to Figure 3, the client's lease begins at t_{C1} and times out at client relative time $t_{C1} + \tau_c$, where τ_c indicates τ counted on the client's clock.

The server acknowledges the clients message at time t_{S2}, and therefore has not begun a lease timeout counter. The server expires the client lease and steals its locks no earlier than $t_{S2} + \tau_s(1 + \delta)$, where τ_s indicates τ counted on the server's clock.

Relatively synchronized clocks imply that $\tau_c < \tau_s(1 + \delta)$. Message ordering implies that $t_{C1} \leq t_{S2}$. We conclude that $t_{C1} + \tau_c < t_{S2} + \tau_s(1 + \delta)$. ■

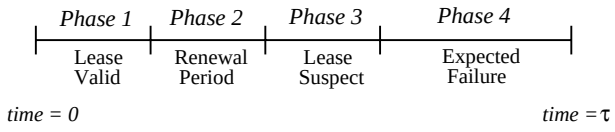

Figure 4. The four phases of the lease period.

A client obtains a new lease on every acknowledged message that it initiates. In this way, clients *opportunistically* renew leases with regular Storage Tank messages to read and write metadata and obtain locks. For regular operation, clients send no additional messages, because the frequency of lock and metadata messages is much higher than the lease interval. Because of opportunistic renewal, the lease protocol incurs negligible overhead. However, we do provide an extra protocol message, with no metadata or lock function, for the sole purpose of renewing a lease. Clients that are no longer actively operating on data, but still cache data and hold locks, use this message to preserve their cache.

3.2. The Lease Interval

The client subdivides the lease period into multiple phases in which it conducts different activities consistent with its lease contract. The client breaks down its lease into four phases (Figure 4). In phase 1, a recently obtained lease protects access to all data objects locked on that server. Local processes request data read and write, and object open, close, get attributes, etc., which the client performs. Any new message ACK received during this period renews the client lease. For this reason, an active client spends virtually all of its time in phase 1.

If a client receives no message acknowledgments in phase 1, often because it is inactive and not initiating communication, the client tries to obtain a new lease. To do this, the client uses a special purpose *keep-alive* protocol message. The keep-alive message encodes no file system or lock operations – it is merely a NULL message requesting an ACK from a server. In phase 2 of the lease period, the client continues to service file system requests on behalf of local processes.

A client that fails to attain a lease in phase 2 assumes that it is isolated from the server. Its active attempts to re-obtain a lease have failed. In phase 3, the client stops servicing file system requests on behalf of local processes. In-progress file system operations continue until the end of the phase. The purpose of this phase, holding back new requests and continuing started operations, is to quiesce file system activity.

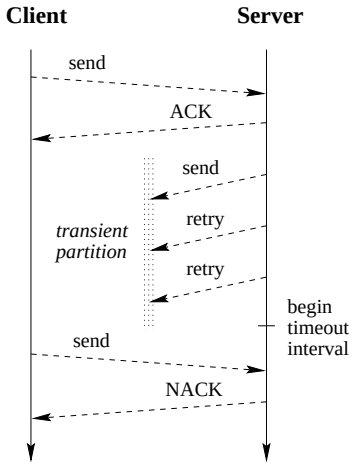

Figure 5. Servers negatively acknowledge messages when timing out clients.

At the end of phase 3, all actions against the files in question are stopped. In phase 4, any dirty data protected by locks associated with the expiring lease are flushed out to disk. By the end of phase 4, no dirty pages should remain. If this is true, the contents of the client cache are completely consistent with the hardened copy written to the storage system.

After the lease expires at the end of phase 4, the lease and its associated locks are no longer valid. The server can steal the client's locks and safely redistribute them to other clients interested in the files. Assuming that the client successfully completed all transactions in phase 3, and finished writing out data in phase 4, sequential consistency for file data is preserved in the presence of a network partition.

3.3. NACKs for Inconsistent Clients

Transient failures and communication errors can result in a client, that believes it is operating on valid leases, communicating with a server that is in the process of running a timer to steal the client's locks. For correct operation, this asymmetry must be detected and addressed in the lease protocol.

For example, a client that experiences a transient network partition misses a message and recovers communication with a server, without knowing it missed the message. The server knows this client to have missed messages and begins timing out the lease, with the knowledge that its cache is invalid. Having recovered communication, the client sends new requests to a server (Figure 5). The server can neither acknowledge the message, which would renew the client lease, nor execute a transaction on the client's behalf. Ignoring the client request, while correct, leads to further unnecessary message traffic when the client attempts to renew its lease in phase 2. Instead, the server sends a negative acknowledgment (NACK) in response to a valid request from a suspect client.

The client interprets the NACK to mean that it has missed a message. It knows its cache to be invalid and enters phase 3 of the lease interval directly. The client, aware of its state, forgoes sending messages to acquire a lease, and prepares for recovery from a communication error.

4. A Comparison with Leases in V

Leases as a synchronization construct were introduced in the **V** operating system at Stanford [11]. While our work draws upon the concepts and semantics of leases in the **V** system, Storage Tank implements a significantly different lease abstraction. In the **V** operating system, a lease represents a period of ownership over a data object, and allows the holder to safely operate on that data object for the lease period. This lease is analogous to a data lock with a built in time-out, and a client holds one lease for every data object that it can write.

Implementing all data locks as leases either introduces a runtime overhead or effects caching policies. Before a lease expires, the holder can renew it, which is required to keep the data in its cache, or purge its cache of that object. The renewal has a message cost, and discarding data effects caching policies.

To reduce the overhead associated with maintaining leases, Storage Tank uses a lease abstraction more congruous with failures in distributed systems. Clients and server stop interacting when the client fails, server fails, or a network partition isolates the two machines. In these cases, all locks held with a single server become invalid. A single lease between each client and server more accurately describes these failures.

5. Related Research

In this work, we have described a protocol for detecting network failures in distributed file systems, and ensuring that caches are consistent when they occur.

Some file systems choose a simple model, where failure detection is not necessary. In the Network File System (NFS) [29], clients poll the server to find out when the file was last modified, and determine whether the cached version is valid. This scheme cannot keep caches coherent. However, it is simple in that servers keep no lock state and do nothing when a failure occurs.

Other file systems choose to steal locks when clients fail. Locks are stolen in the fashion described in Section 1.2. Examples include the Andrew file system [15], Sprite [4], and the DEcorum file system. Since these file systems marshal all I/O requests through a server, stealing locks is safe.

In file systems that are peer based, and have no server, or use direct access to network addressable storage, stealing locks is not safe. A prevalent and successful alternative to

stealing locks is to implement a file system in conjunction with group management and replication software that performs failure detection and manages consistent distributed data views. For these systems, safe caching is a by-product of the safe computing environment. Examples include: the Deceit file system [26] built on top of the Isis computing environment [5]; and, the Calypso file system [7] implemented on another group service [14].

Other file systems built on replication services take an optimistic approach, and allow unsynchronized concurrent updates to data. Conflicting updates to data must be detected and repaired as best as possible. These systems choose high availability before strong cache consistency. Examples include the CODA file system [25], the LOCUS distributed file system [28] and the Ficus replicated file system [12].

The solutions most closely related to ours are other timeout or lease based locking schemes. The Global File System [23] is also a SAN file system. To synchronize actions between clients they use the dlock construct, implemented by the disk drive, which locks a range of disk addresses. These locks have timeout counters, enforced by the disk drive, so that they are available to other file system clients after failure. In Storage Tank, locking is logical, locking distributed data structures, rather than physical, locking a disk address range. For this reason, we feel that the dlock alone is not adequate for our distributed file system.

The Frangipani file system [27] uses a lease most similar to Storage Tank's. Each computer holds a single lease with a locking authority and the lease protects cached data until it expires. Frangipani uses heartbeats and loosely synchronized clocks between the locking authority and computers, rather than ordered events and rate synchronized clocks. Also, Frangipani stores lease information at the locking authority, rather than having a passive authority.

6. Comments and Future Directions

This work addresses cache coherency in the presence of network failures, server failures and client failures. Noticeably absent is a discussion of availability, consistency, and recovery at the server from server failure. Distributed file servers, like Storage Tank, that maintain lock and client state must recover that state after a server failure. File systems can do this through replication [26, 12, 27], replication with hardware support [13], client-driven lock reassertion [16, 7, 4], server polling of client state [2, 4], and by maintaining non-volatile state at the server [4]. Storage Tank uses a combined policy of lock reassertion and hardware supported replication. For this work it is assumed that Storage Tank servers are highly available and recover from failures.

One of the assumptions in the lease-based safety protocol is that clocks are rate synchronized, which implies that computers do not exhibit partial failure by executing commands slowly. This assumption is reasonable for most computers, which have hardware clocks that continue to operate even when the computer's software has failed, but cannot necessarily be guaranteed by the network and storage subsystems. To address slow computers, we use fencing in addition to the lease protocol. At the same time the server times-out a client's locks, it constructs a fence between that client and its storage devices. The fence prevents late commands, from a slow computer, from accessing the disk after locks are stolen. While fencing cannot guarantee data consistency, it can prevent unsynchronized conflicting accesses that the lease-based protocol does not detect.

Analytical results [11] and experimental results [27] show that leasing has little impact on system performance. With our improved lease semantics, opportunistic renewal and lazy server evaluation, we feel strongly that leases will have little effect on performance in Storage Tank. While lease performance is well accepted in file system research, measurement of modern file system workloads are required to experimentally verify our design. A next step in our research is to validate our leasing design on a Storage Tank prototype.

7. Conclusions

The emergence of network addressable storage, like SANs, makes it more difficult for a file system to provide high availability in the presence of failures. In particular, stealing locks, the accepted solution for recovering from client failures, no longer provides adequate cache coherency and data consistency. By implementing a data safety protocol in a distributed system using leases, Storage Tank protects the consistency of data at little or no runtime cost. This approach addresses cache coherency as well as data consistency, even for isolated computers, allowing them to report errors correctly.

Storage Tank reduces the overhead associated with a lease-based safety protocol by both obtaining leases opportunistically, and requiring no server processing during regular operation. Leases, which were shown to have little impact on server and network performance [11] in the **V** operating system, should perform even better given these optimizations.

References

[1] A. Acharya, M Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 1998.

[2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file

systems. *ACM Transactions on Computer Systems*, 14(1), February 1996.

[3] O. Babaoglu, R. Davoli, L. A. Giachini, and M. G. Baker. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995.

[4] M. L. G. Baker. *Fast Crash Recovery in Distributed File Systems*. PhD thesis, University of California at Berkeley, 1994.

[5] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985.

[6] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4), 1991.

[7] M. Devarakonda, D. Kish, and A. Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3), August 1996.

[8] C. Dwork, C.-T. Ho, and R. Strong. Collective consistency. In *Proceedings of the Distributed Algorithms 10th International Workshop*, 1996.

[9] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Performance Evaluation Review*, volume 25, 1997.

[10] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attach secure disks. Technical Report CMU-CS-97-118, School of Computer Science, Carnegie Mellon University, July 1997.

[11] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[12] R. G. Guy, J. S. Heidemann, W. Mark, Jr. T. W. Page, G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *Proceedings of the 1990 Summer Usenix Conference*, 1990.

[13] J. H. Hartman and J. K. Ousterhout. The Zebra-Striped network file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*. ACM, 1993.

[14] F. Jahanian, R. Rajkumar, and S Fakhouri. Processor group membership protocols: Specification, design, and implementation. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1993.

[15] M. L. Kazar. Synchronization and caching issues in the Andrew file system. In *Proceedings of the USENIX Winter Technical Conference*, February 1988.

[16] M. L. Kazar, B. W. Leverett, O. T. Anderson, V. Apostolides, B. A. Bottos, S. Chutani, C. F. Everhart, W. A. Mason, S. Tu, and R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer USENIX Conference*, June 1990.

[17] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD*, 27(3), September 1998.

[18] L. Lamport. Time, clocks and the ordering of events in a distributed systems. *Communications of the ACM*, 21(7), July 1978.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), 1979.

[20] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), March 1986.

[21] K. Muller and J. Pasquale. A high performance multi-structured file system design. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[22] J. Palmer, R. Strong, and E. Upfal. Nonblocking membership protocols with asymmetric safety. Technical Report RJ–10096 (91912), IBM Research Division, December 1997.

[23] K. W. Preslan, A. P. Barry, J. E. Brassow, G. M. Erickson, E. Nygaard, C. J. Sabol, S. R. Soltis, D. C. Teigland, and M. T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the 16th IEEE Mass Storage Systems Symposium*, 1999.

[24] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th International Conference on Very Large Databases*, 1998.

[25] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), April 1990.

[26] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. In *Proceedings of the 1990 Summer Usenix Conference*, 1990.

[27] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, 1997.

[28] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983.

[29] D. Walsh, B. Lyon, G. Sager, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In *Proceedings of the 1985 Winter Usenix Technical Conference*, January 1985.