Semi-preemptible locks for a distributed file system

Conference Paper · March 2000

DOI: 10.1109/PCCC.2000.830343 · Source: IEEE Xplore

Randal C. Burns
Johns Hopkins University
255 PUBLICATIONS **10,082** CITATIONS

Darrell D. E. Long
University of California, Santa Cruz
307 PUBLICATIONS **8,712** CITATIONS

Semi-Preemptible Locks for a Distributed File System

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@cse.ucsc.edu

Abstract

Many applications require the ability to obtain exclusive access to data, where an application is granted privileges to an object that cannot be preempted and limits the actions of other processes. Local file systems support exclusive access to files by maintaining information about the access rights of current open file instances, and checking subsequent opens for compatibility. Implementing exclusive access in this manner for distributed file systems degrades performance by requiring every open file to be registered with a server that maintains global open state.

We introduce a distributed lock for managing file access, called a *semi-preemptible* lock, that avoids this performance limitation. Clients locally grant open requests for files that are consistent with a distributed semi-preemptible lock that they hold. File system clients retain or cache distributed locks, even in the absence of open file instances. When a file access lock is already cached, a client services open requests without a server message, improving performance by exploiting locality, the affinity of files to clients.

1 Introduction

A distributed client-server file system presents a local file system interface to remote and shared data. The file system client takes responsibility for implementing the semantics of the local file system and translating the local interface onto a client/server network protocol. For heterogeneous distributed file systems (concurrently running on many client operating systems), the system may be translating the semantics of several different local file systems onto a single network protocol.

Distributed file systems have become the principal method for sharing data in distributed applications. Programmers understand local file system semantics well, and use them to easily gain access to shared data. For exactly the same reason that distributed file systems are easy to use, they are difficult to implement. The distributed file system takes responsibility for providing synchronized access and consistent views of shared data, shielding the application and programmer from these tasks, but moving the complexity into the file system.

In this work, we present a locking construct, *file access locks*, that are used to implement the semantics of file open, *e.g.* if one client opens a file for exclusive writing, permitting no shared readers, concurrent opens for read on other clients must be forbidden. These locks are not designed to provide data consistency or cache coherency – a suitable cache coherency protocol is required in addition to file access locking. Instead, the locks allow clients to choose from among the many exclusive access and sharing options available in its native file system interface, and have the semantics of the local open enforced throughout a distributed system.

Traditionally, distributed file systems have relaxed local semantics for ease of implementation and performance reasons. For example, the Andrew file system (AFS) [13] chooses a considerably simpler model for synchronizing file access than the POSIX local file system interface of its clients. AFS does not have a distributed mechanism to enforce exclusive open modes between clients. By relaxing semantics, AFS has the ability to aggressively cache data after a file has closed. This provides high performance, low latency access to data on subsequent opens.

Some file systems, like the CIFS network file system [17], are willing to suffer network message overhead in order to correctly enforce local semantics. These file systems send all open and all close operations to a server. The server maintains a globally consistent view of file system open state, and checks every open request against all outstanding open instances. We view this as an unacceptable solution due to the limitations that it places on clients. Registering open and close requests means that a client conducts connection or session oriented transactions with servers. For CIFS, this prevents clients from caching file data when the file is not currently open. CIFS has subsequently improved performance for opens by providing what they call an opportunistic lock, allowing a single client to cache data and access privileges past close. However, opportunistic locks apply to limited and specific

0-7803-5979-8/00 $10.00 © 2000 IEEE

cases of data sharing, and are less general than our locking system.

Our contribution to synchronized file access in the distributed environment is the *semi-preemptible* lock. A client holding a semi-preemptible lock on a file has the right to access a file in any of the modes specified by its held lock. Clients maintain their own file open state locally, and do not need to transact with the file system server when opening or closing file data. Clients may continue to hold such a lock even when they have no open instances. In this way, a client can cache access privileges, the right to open a file, and service subsequent open requests without a message to the server. This mechanism reduces server traffic by eliminating open and close messages, and consequently reduces latency by avoiding message round trip time. Clients cache access privileges to a file on the belief that the file will be used again locally before being used by another client. The same temporal locality that makes data caching effective in the distributed environment [19, 14] is used to improve performance for file open.

Semi-preemptible locks also reduce distributed lock state. Clients often hold multiple open instances of a single file concurrently. All of these open instances can be granted under a single semi-preemptible lock, rather than holding a separate lock for every open. A single semi-preemptible lock summarizes all of the client's open state to the distributed system.

The semi-preemptible lock avoids client-server interactions when compared to existing protocols for synchronization of file open and close requests in a distributed system. Through the caching of locks, subsequent opens of a file on the same client can be served locally.

2 A Storage Area Network File System

A brief digression into the file system architecture in which we implement semi-preemptible locking helps to motivate the performance advantages.

In the *Storage Tank* project at IBM research, we are building a distributed file system on a storage area network (SAN) (Figure 1). A SAN is a high speed network that gives computers shared access to storage devices. Currently, SANs are being constructed on Fibre Channel (FC) networks [3]. In the future, we expect network attached storage devices to be available for general purpose data networks, so that SANs can be constructed for networks such as Gigabit Ethernet [7]. A distributed file system built on a SAN removes the server bottleneck for I/O requests by giving clients a direct data path to disks.

When building a distributed file system on a SAN, file system clients can access data directly over the storage area network. Most traditional client-server file systems

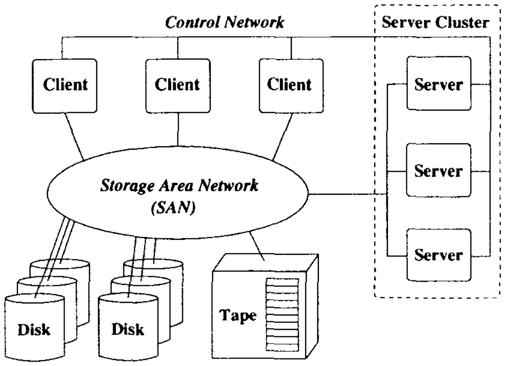


Figure 1: Schematic of the Storage Tank distributed file system on a storage area network (SAN).

[24, 13, 15, 6] 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. This direct data access model is similar to the file system for network attached secure disks (NASD) [9], using shared disks on an IP network, and the Global file system [20], for SAN attached storage devices.

Clients communicate with Storage Tank servers over a general purpose network to obtain file metadata. In addition to serving file system metadata, the servers manage cache coherency protocols, authentication, and the allocation of file data (managing data placement and free space).

Unlike most file systems, metadata and data are stored separately. Metadata, including the location of the blocks of each file on shared storage, are kept on high-performance storage at the server. The SAN storage devices contain only the blocks of data for the files. 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. Previously, server performance was measured by data rate. Performance was occasionally limited by network bandwidth, but more often limited by the server's ability to read data from storage and write it to the network. In the SAN environment, without any storage or network I/O, 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. By using computational resources more efficiently, Storage Tank should present a

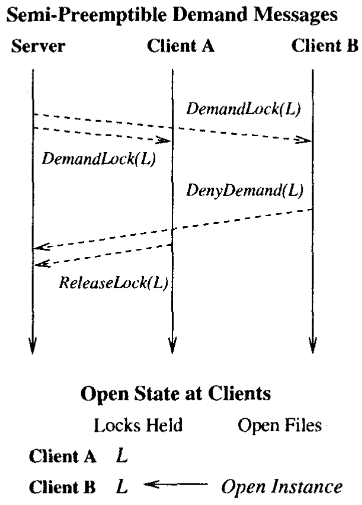


Figure 2: Demands for the semi-preemptible lock are accepted or denied depending upon client file system open state.

higher data rate to the file system client, and allow a distributed file system to fully utilize its network infrastructure with less investment in server hardware.

Without the relatively slow process of shipping data from the client to the server to hide protocol overhead, minimizing the message traffic for file system operations becomes important. Protocol overhead is the added latency, and network and server resources used for client-server messages. In traditional client-server file systems, clients go to the server to obtain data. Because shipping data to the client takes significantly longer and uses many more resources than a single server message, the overhead associated with the messages for opening and closing a file are hidden by the cost of shipping data. In Storage Tank, protocol overhead limits performance. When the semi-preemptible lock allows a client to open a file without contacting the server, a message is avoided and time is saved on the server's critical path.

3 Semi-Preemptible File Locks

For distributed files systems, little data sharing occurs in practice [2, 16], where data sharing indicates two clients concurrently accessing the same file. Additionally, clients often access data that they have recently used. These claims are supported by the effectiveness of data caching in this environment [19, 14]. More mature distributed file systems [13, 6, 22, 15, 1, 12, 19, 21] take advantage of this observation and cache file data at clients even when no process actively uses the data. The design decision to cache file data after a file has been closed improves performance when a subsequent open from the same system is more likely than a request from another client to access the same data. For subsequent accesses from the same system, caching avoids a server message and data read from disk, and performance improves. However, if another client attempts to access the same data, it sees additional latency while the file system server invalidates the cache of the client that holds data before granting access to the new client.

For the same reasons that caching improves performance on data access, the semi-preemptible lock improves performance on file open. When a local process requests an open for a file, if no current lock is held, the client obtains a semi-preemptible file access lock before granting the open. When the local process closes the file, the client records that there are no open file instances currently using the lock, but holds onto the lock awaiting future opens of the same file. Analogous to data caching, holding access locks past close decreases latency by avoiding a server message when the same client attempts to open the file. But, caching locks hinders opens from other clients, because the server must demand the lock before granting access to the other client.

By recording the open instances associated with each access lock, a client differentiates locks that are held to protect open files, and therefore cannot be released, from locks that are held to improve performance on subsequent opens. Consider that a second client wishes to obtain exclusive access to a file that is already locked by a first client. The server processes the second client's request by *demanding* [1] the lock from the first client, sending a message that requests the release of the held lock. If the first client has a process that holds an open instance of that file, it requires the held lock to protect that open instance and denies demand requests from the server (Figure 2). However, if no process holds an open instance of the file, the client no longer utilizes the held lock and releases it safely. These access locks are called semi-preemptible, because the server must demand them, as if they were preemptible locks. However, a client can refuse a demand request.

The semi-preemptible locking system is particularly appropriate for our SAN-based distributed file system, because data are not obtained through the server. When a file access lock is held, a client can directly access the data from shared storage, and need not interact with the server at all. This claim is also true for other direct access storage environments, like network attached secure disks (NASD) [8]. Without direct access storage, clients must interact

[1] We use the term "demand" for consistency with existing lock terminology despite the fact that clients can refuse a demand. A term like "request" would more appropriately describe the server's action, but request is often used to describe the client's process for acquiring locks from the server.

Lock	Name	Access	Sharing
M	Metadata	$\mathcal{M}$	$\mathcal{M}, \mathcal{R}, \mathcal{W}$
R	Read	$\mathcal{M}, \mathcal{R}$	$\mathcal{M}, \mathcal{R}, \mathcal{W}$
S	Shared	$\mathcal{M}, \mathcal{R}$	$\mathcal{M}, \mathcal{R}$
W	Write	$\mathcal{M}, \mathcal{R}, \mathcal{W}$	$\mathcal{M}, \mathcal{R}, \mathcal{W}$
U	Update	$\mathcal{M}, \mathcal{R}, \mathcal{W}$	$\mathcal{M}, \mathcal{R}$
X	Exclusive	$\mathcal{M}, \mathcal{R}, \mathcal{W}$	$\mathcal{M}$

Table 1: The MRSWUX locks are combinations of protected access modes and allowed sharing modes selected from a metadata ($\mathcal{M}$), read ($\mathcal{R}$), and write ($\mathcal{W}$) access.

with servers for data, and these file systems cannot save a message from semi-preemptible locking.

4 Managing File Access

To enforce open semantics in a distributed system, we define a suitable set of file access locks and a mapping of local file system open modes to these access locks. In certain cases, the local file system open modes for our client platforms cannot be paired to a distributed file access lock with the same semantics. For these exceptions, we map unsupported open modes to more restrictive file access locks. This mapping always guarantees that the lock holder has the requested privileges, but reduces the ability of other clients to hold locks concurrently. While the chosen locks do not exactly implement open mode semantics for all client file systems, they do guarantee to keep consistent views of distributed open state across all clients. Deviations from strict local semantics in Storage Tank are minor, and an improvement over the file access synchronization provided in all known distributed file systems [6, 14, 15, 4]. The designers of Storage Tank believe that these locks describe an intuitive and descriptive data sharing model.

4.1 The MRSWUX Locking Scheme

Table 1 describes the semantics of the defined locks. Distributed file system clients select from this table the lock that describes their intended access, and the access modes of other open instances with which they are willing to share. Our system defines three access modes: read access, allowing the holder to open files for read and data caching; write access, permitting the holder to open files for write and write caching; and metadata access, the right to cache and read metadata.

Client file systems enforce that local processes do not open a file in a mode that exceeds the privileges granted in the intended access field or restricts sharing more than the granted lock.

In our distributed access model, we have decided that any lock that can write data may also read data, and that any lock mode that shares access with writers also shares access with readers. This read/write hierarchy appears in most protocols for implementing consistency in a distributed system [5].

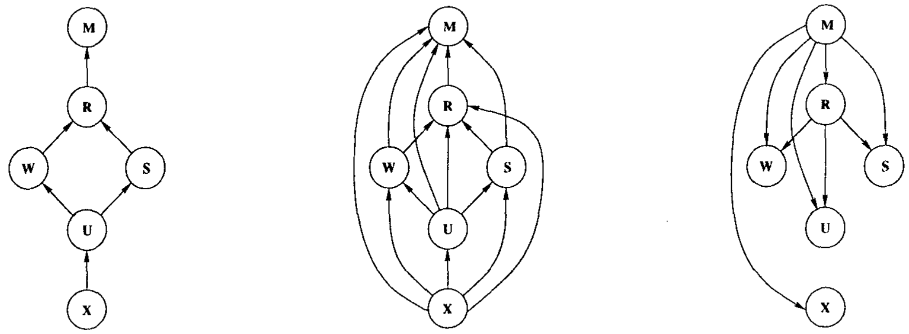
	Held Lock					
Requested Lock	M	R	S	W	U	X
M	+	+	+	+	+	+
R	+	+	+	+	+	−
S	+	+	+	−	−	−
W	+	+	−	+	−	−
U	+	+	−	−	−	−
X	+	−	−	−	−	−

Table 2: Lock compatibility table.

Unlike distributed data systems, local file systems often separate read and write access, choosing a flexible interface, rather than one congruent with the semantics of data sharing. Local file systems find it useful to differentiate between read/write and write-only to improve the performance of local data cache management strategies and to differentiate special device files, such as pipes, with limited access. While reasonable for local file systems, these concepts do not apply to distributed file sharing and are left out of MRSWUX locking.

From these locking modes, the associated compatibility table [11] (Table 2) describes the pairwise combinations of locks that are compatible. A distributed file system server uses this data structure to resolve what action to take when a client requests a semi-preemptible lock. The server evaluates the requested lock (row) against all outstanding held locks (column). The + marks in the table indicate that two lock modes are compatible. The − marks denote incompatibility. When evaluating a lock request against held locks, the server either grants it when compatible, or when incompatible attempts to acquire the requested lock on behalf of the client by demanding all incompatible locks. If a lock demand is denied by a client, indicating that a held lock protects a current open instance, the server denies the requested lock.

When using semi-preemptible file access locks, the server has no knowledge of the global open state of a given file. Clients maintain this state individually. The server must therefore forward lock requests to be resolved at clients through the semi-preemptible mechanism. Distributed file access locks indicate to the system the set of clients that potentially hold a file open. The server queries these clients to resolve conflicts. Our system does not contain a centrally located global record of open files, but lock demands provide an equally descriptive method of determining the global open state of a file.



(a) Hierarchy (b) Downgrades (c) Upgrades

Figure 3: Graphs for the MRSWUX locking scheme.

4.2 Summarizing Local Open State

In Storage Tank, we restrict each client's lock holdings to a single distributed file access lock for each file. Clients may have many local open instances of that given file protected by a single lock. This further reduces the number of held locks in a distributed system and decreases network traffic associated with file locking. To make this restriction possible, the set of locks we define must form a *compatible hierarchy* and we must provide a facility for clients to change their lock holdings without releasing locks.

To ensure that a single distributed file access lock always suffices to describe multiple local open instances, the set of defined locks must form a compatible hierarchy (Figure 3(a)). A suitable hierarchy meets the condition that all compatible lock are *strength* related. Strength indicates that the *stronger* lock protects all access modes allowed by the *weaker* lock, and the stronger lock does not allow any locks to be concurrently held that conflict with the weaker lock. In other words, a stronger lock allows the lock holder more file access modes and shares fewer. Note that the only locks that are not strength related (S and W) are not compatible.

Locks that form a compatible hierarchy allow file opens to be granted under any lock of equal or greater strength than the minimum strength lock to which the file open can be mapped. Also, it permits multiple compatible opens to be protected by a single lock, of equal or greater strength than lock required to protect the strongest open.

Clients using the MRSWUX locking scheme must be able to modify their currently held lock – change the protected access and sharing without releasing the lock. This need arises in two instances: 1) when the client holds a semi-preemptible lock protecting local open instances and another client requests a lock for that file that does not conflict with the open instances, but is incompatible with the held semi-preemptible lock; and 2) when a client holding a semi-preemptible lock that protects open instances has a local process request another open instance, compatible with current open instances, that has access and sharing requirements the semi-preemptible lock cannot provide.

In the first case, the held lock is too strong and the client must convert it to a weaker and compatible lock that still protects the open instances. This process is a lock *downgrade*. The client cannot release its lock outright and obtain a weaker lock because it has current open instances to protect. For downgrades, the lock demands received from a server must contain the type of file access lock requested so that clients can resolve compatibility and the appropriate downgrade.

The legal downgrade graph (Figure 3(b)) is constructed by taking the transitive closure of the hierarchy graph, and shows all transitions from a stronger to a weaker lock. A downgrade can always be performed; for any held lock, all locks weaker than the held lock are compatible with the system's other outstanding locks.

In the second case, the held lock cannot provide the access and sharing requirements of the new request. The held lock is too weak and the client attempts to obtain a stronger lock that can protect all of the current open instances and the new request. Obtaining the stronger lock while continuing to hold the old lock is called a lock *upgrade*.

The legal upgrade graph (Figure 3(c)) shows all transitions from held locks to compatible stronger locks. Limiting upgrades to compatible locks prevents the locking system from falsely refusing legal lock transitions. We discuss this topic in depth in Section 4.3.

Unlike compatibility, which determines whether pairs of locks can be held concurrently, the concepts of strength and weakness help the system determine what action to take, *i.e.* how to change lock state to service local open

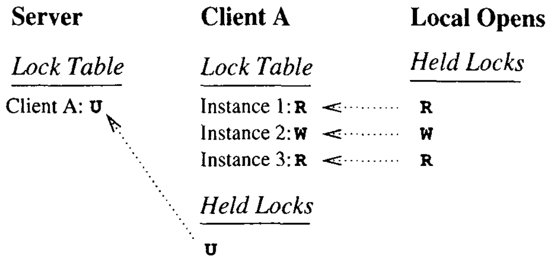


Figure 4: Distributed and local file access locks.

requests or server demand messages.

4.3 Lock Management

The client uses the concept of compatibility and lock strength to grant local opens under the protection of one distributed lock. Each local open instance can be represented by the minimum strength lock from MRSWUX that provides the requested access modes and limits concurrently held locks. For each local open instance, the client assigns it a *local lock* (Figure 4). The local lock is used only to evaluate compatibility, and upgrades and downgrades. The client manages these local locks so that 1) for any file all local locks are compatible with each other, and 2) the client holds a distributed lock stronger than or of equal strength to all local locks. For a client that enforces these conditions, the held distributed lock encapsulates its open state.

The client might not be using all of the strength of its held file access lock, but if a server acts in accordance with the held lock, the server makes no illegal actions. Recall that the server demands the held lock to try and reduce its strength if an incompatible request arrives from another client. Therefore, holding a stronger lock does not impact correctness.

As previously stated, a client never needs to upgrade from a held lock to a stronger incompatible lock. Upgrade requests occur when a local file open requires a stronger lock than the held lock. If the client needed the full strength of the current lock, it would have an open instance incompatible with its new open request. In cases where the client needs an upgraded lock incompatible with the currently held lock, it is not using the full strength of the held lock. Clients address this situation by downgrading their current holding to the minimum strength lock that protects current local locks before upgrading to the needed lock. An example in our MRSWUX locks clarifies this point. Suppose that a client holds an W or S lock that protects a local open using a local R lock. If the client receives another open request that requires a local U lock. The request is compatible with the local lock, but incompatible with the held distributed file access lock. The client downgrades its current holding to R and then upgrades to U. In limiting upgrades to compatible locks only, we avoid subtle race conditions that lead to unnecessary demands.

dwDesiredAccess	Intended Access
0	Metadata access only
GENERIC_READ	Read
GENERIC_WRITE	Write
DELETE	Deletion

(a) Intended access mode

dwSharingMode	Compatibility
0	None
SHARE_READ	Any reader
SHARE_WRITE	Any writer
SHARE_DELETE	Deletion

(b) Sharing mode

Table 3: Sharing modes in Windows-NT.

5 Windows-NT – A Case Study

The presented locking scheme intends to support local file system semantics with a simplest design and as few locking states as possible. MRSWUX locking allows a distributed file system to exactly implement POSIX semantics. However, due to the small number of locks for MRSWUX, the locks do not cover all Windows-NT open modes correctly, and violations of the Windows-NT semantics are possible.

Table 3 presents the options to the Windows-NT function call CreateFile used to open a file. The dwDesiredAccess argument describes the access mode that the open protects. Similarly, the dwSharingMode describes the access modes that can be legally held by concurrent open instances. All combinations of these arguments are legal. Based on the arguments to CreateFile, we outline the key shortcomings of MRSWUX when used to implement a Windows-NT file system client for Storage Tank.

When mapping the CreateFile arguments to a lock from MRSWUX, we ignore the DELETE and SHARE_DELETE option as it applies to file deletion and not to file sharing. For file access options, we select the minimum strength lock that has the desired access and sharing modes. This mapping is exact except when the SHARE_WRITE appears without SHARE_READ or GENERIC_WRITE without GENERIC_READ. In these cases, we include a read privilege even though it was not requested, *e.g.* arguments GENERIC_WRITE, SHARE_WRITE obtains a W lock which includes an unrequested read privilege and read sharing.

MRSWUX does not differentiate between open with intent to write and open with intent to both read and write. While we justified this decision by citing that it is not significant for file sharing, the distinction is part of Windows-NT semantics. The chosen locks fail to adequately describe these options and may fail to grant concurrent opens on separate clients that file system semantics allow. For example, Windows-NT semantics permits two clients to concurrently open files for GENERIC_WRITE,SHARE_WRITE, whereas our system would prohibit this open to occur.

To mitigate the effects of inexact mappings, our system does allow two clients to both open a file for GENERIC_WRITE,SHARE_WRITE as long as they are on the same system. For concurrent opens, existing local synchronization semantics override distributed locking. Consequently, semantic violations only occur between distinct clients. Local correctness for single clients mitigates the shortcomings of the MRSWUX locking scheme by taking care of the most common occurrences of file sharing.

By not implementing Windows-NT open semantics exactly, MRSWUX can prevent multiple clients from opening a file concurrently that local semantics would allow on a single client. This can only occur when clients request write access without a read. MRSWUX never allows opens that a local semantics forbid.

Despite the shortcomings, we feel that MRSWUX locks effectively manage exclusive file access in the Windows-NT environment. We choose to implement these locks for their simplicity. Implementing exact semantics locally addresses the most common cases of sharing, and the semantic violations influence concurrency, not correctness. The key feature of MRSWUX is the reduction of complexity, supporting many local file system open modes with few locks and small locking data structures.

6 Related Research

Many distributed file systems clients transact with a server on every open and close of a file for synchronization [23, 4, 24]. This is the simplest technique to implement local file system open semantics in the distributed environment. However, all file open and file close requests require a network operation.

The Andrew file system [14] interacts with a server at every open and close and uses open and close as points to synchronize cached data. Andrew does implement a data cache that can hold data past close and a general *callback* (demand) mechanism. However, callbacks apply only to preemptible data locks.

Like Storage Tank, the Calypso file system [6] uses open mode synchronization locks, called tokens, to implement local file system semantics. In Calypso, tokens are fully preemptible at the client and cannot be held past close. Every file system open and close generates a token request or release. File system open state and token request conflicts are managed completely at the server. Calypso uses a simple lock hierarchy for its data locks, but the hierarchy does not apply to open mode synchronization tokens.

The DFS file system [15] describes a token mechanism similar to semi-preemptible locks for the management of data, metadata, and open state. Like semi-preemptible locks, a client can refuse or permit revocation of a token, depending upon local state. Token management in DFS differs in that all elements of system locking, including file access locks, data locks, and byte-range locks are managed with the single token mechanism. The DFS treatment of token management is less concrete than our discussion of locking, and does not address mapping local file system semantics to a distributed locking system.

7 Comments and Future Directions

Our argument that semi-preemptibility and client lock summarization improve performance relies on temporal locality and limited data sharing between clients. Nearly identical arguments were made for caching in the Andrew file system [14] and hoarding for Coda's disconnected operation [16]. While these arguments are well accepted in file system research, we feel that simulation and measurement of modern file system workloads are required to experimentally verify our design. A next step in our research is to validate our locking design on a Storage Tank prototype and quantify performance improvements.

For space reasons, this work omits a discussion of synchronization in the presence of failure. A distributed file system that presents a local file system interface to remote and shared storage must continue to do so when components fail. As do many modern file systems [22, 20, 18], Storage Tank uses a lease-based [10] protocol to ensure operational safety and high availability in the presence of client and server failures, and network partitions.

8 Conclusions

Distributed file systems need to manage open state for referential integrity and synchronized access to files. Existing distributed systems address this problem by either relaxing local file system semantics, or by sending every file open request to a server. We have introduced a locking construct, the semi-preemptible lock, that permits file system clients to grant most file open requests locally, without a server transaction. By avoiding server messages on open, our client improves performance by exploiting locality of access to files.

At the file system client, semi-preemptible locks are used to summarize open state, so that many open files may be granted under the protection of a single semi-preemptible lock. This reduces global lock state and further reduces client-server messages.

Distributed file systems play a central role in distributed computing because they mask the complexity of distributed data management, and users understand the file system interface. With semi-preemptible locking, we continue to approach the goal of providing high performance file access in a distributed system.

References

[1] 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):41–79, February 1996.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Annual Symposium on Operating Systems*, October 1991.

[3] A. F. Benner. *Fibre Channel: Gigabit Communication and I/O for Computer Networks.* McGraw-Hill Series on Computer Communications, 1996.

[4] A. D. Birrell and R. M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, SE-6(5), September 1980.

[5] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, USA., 1999.

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

[7] H. Frazier and H. Johnson. Gigabit ethernet: From 100 to 1,000 Mbps. *IEEE Internet Computing*, 3(1), 1999.

[8] 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.

[9] 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.

[10] 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.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, California, USA., 1993.

[12] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

[15] 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.

[16] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), 1992.

[17] P. J. Leach. A common Internet file system (CIFS/1.0) protocol. Technical report, Network Working Group, Internet Engineering Task Force, December 1997.

[18] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2), May 1994.

[19] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1):137–152, February 1988.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] D. Walsh, B. Lyon, G. Sager, J. Chang, D. Goldberh, 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.