Cluster Computing 4, 295–306, 2001

# Scalable Session Locking for a Distributed File System

RANDAL C. BURNS and ROBERT M. REES
*Storage Systems and Software, IBM Almaden Research Center, USA*

LARRY J. STOCKMEYER
*Department of Computer Science, IBM Almaden Research Center, USA*

DARRELL D.E. LONG
*Department of Computer Science, University of California, Santa Cruz, USA*

**Abstract.** File systems provide an interface for applications to obtain exclusive access to files, in which a process holds privileges to a file that cannot be preempted and restrict the capabilities of other processes. Local file systems do this by maintaining information about the privileges of current file sessions, and checking subsequent sessions for compatibility. Implementing exclusive access in this manner for distributed file systems degrades performance by requiring every new file session to be registered with a lock server that maintains global session state. We present two techniques for improving the performance of session management in the distributed environment. We introduce a distributed lock for managing file access, called a *semi-preemptible* lock, that allows clients to cache privileges. Under a semi-preemptible lock, a file system creates new sessions without messages to the lock manager. This improves performance by exploiting locality – the affinity of files to clients. We also present data structures and algorithms for the *dynamic evaluation* of locks that allow a distributed file system to efficiently manage arbitrarily complex locking. In this case, complex means that an object can be locked in a large number of unique modes. The combination of these techniques results in a distributed locking scheme that supports fine-grained concurrency control with low memory and message overhead and with the assurance that their locking system is correct and avoids unnecessary deadlocks.

**Keywords:** concurrency control, distributed file systems, session locking, lock evaluation

## 1. Introduction

Distributed file systems have become the principal method for sharing data in distributed applications. Programmers understand 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. 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 (many client operating systems), the system may be translating the semantics of several different local file systems onto a single network protocol.

In this work, we present a locking construct, called *file session locks*, that implement sessions on files defined by open and close calls from an application. File sessions enforce concurrency constraints; e.g., if one client opens a file for exclusive writing, permitting no concurrent readers, 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 session 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.

To help encode and enforce sessions in a distributed environment, we contribute the *semi-preemptible* lock, which allows file system clients to cache session privileges. 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 (session) state locally, and do not need to transact with the file system server when opening or closing a file. 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.

Semi-preemptible locks also reduce distributed lock state. Clients often hold multiple open instances of a single file concurrently. Clients locally create concrete locks for each session which are held under (or in the context of) a distributed lock. 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.

Semi-preemptible locks combine two concepts in lock management. First, clients manage locks *hierarchically* to separate session state from lock state. Second, clients use semi-preemptibility, also called lazy-revocation or sticky locks [23], to retain privileges on files in the absence of active sessions. We use the term hierarchical differently than most of the locking literature. Generally, the term hierarchical describes locks held concurrently on different levels of abstraction on the same data object to achieve granular locking [11,26]. Instead, we use the term to describe two levels of locking – the higher level is abstract, summarizing client lock state to the server, and the lower level is concrete, representing sessions.

The interface to open a file (create a session) has many options that allow an application to specify its intended actions and restrict concurrent actions by other clients. To express these options, locks on these data generally have multiple locking *modes* each with unique semantics. The number of different possible locking modes increases exponentially with the number of access methods and locking for a complex data object using existing methods quickly becomes unmanageable. We take the Windows-NT interface as an example to establish this point. The file system takes six binary arguments when opening a file which requires 64 unique locking modes to describe all sessions. Considering locking modes pairwise for *compatibility*, there are 4096 combinations. Locks are compatible if they can be held concurrently. Furthermore, the large number of modes in this interface do not fully specify all possible access modes. For example, they do not address concurrent or exclusive access to file system metadata.

Existing methods for lock management fall short in that they either fail to scale well as the number of locking modes become large, or they are ad hoc systems that are poorly specified. Database systems employ a static data structure, called a lock compatibility table [11], that describes the relationships between all locking modes pairwise. This data structure fully specifies the interactions between locking modes, but grows quadratically in size with respect to the number of locking modes. Alternatively, modern file systems use ad hoc rule-based methods to evaluate locks that are efficient and compact [15]. However, without formally specified semantics, the interactions between locking modes can be difficult to reason about and implement correctly.

We present a formal specification of locking modes and derive from this a data structure and algorithms for the management of distributed locks called dynamic evaluation. Our methods have the advantage of scaling well with the number of locking modes. In addition, they specify fully the interactions among all locking modes, which eases the implementation of a correct locking protocol. Our system obviates the need for static data structures, such as the lock compatibility tables used in database systems, which require memory in $\Omega(n^2)$ with respect to the number of legal locking modes. In our system, lock compatibility is evaluated algorithmically based on small static lock structures that require memory in $O(\log n)$. While, a distributed file system provides a good example of a complex locking system, we feel the dynamic lock evaluation has wide applicability to distributed systems.

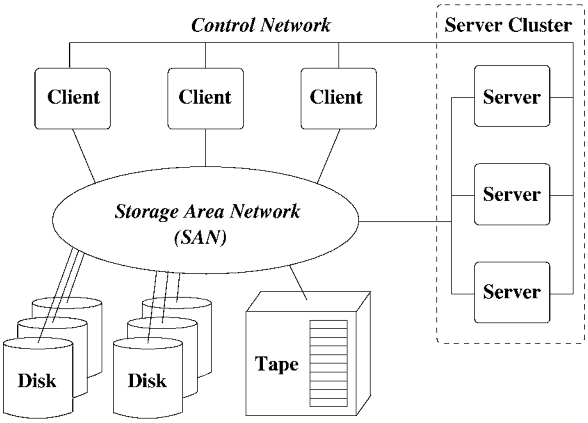

Figure 1. Schematic of the Storage Tank distributed file system on a storage area network (SAN).

## 2. A storage area network file system

A brief digression into the file system architecture in which we implement file session 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 [4]. 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 [9]. 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.

File system clients on a SAN access data directly over the storage area network. In contrast, most traditional client–server file systems [7,13,15,25] store data on the server's private disks. Clients 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) [10], 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 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 a 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, 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 server performs many more transactions than a traditional file server with equal processing power.

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 network and server resources and added latency 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 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 [14,18]. More mature distributed file systems [1,7,12,13,15,18,22,23] 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, caching improves performance by avoiding a server message and data read from disk. 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 lock is held, the client obtains a semi-preemptible 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 on subsequent opens. But, caching locks adds latency to opens from other clients, because the server must revoke the lock before granting access to the other client.

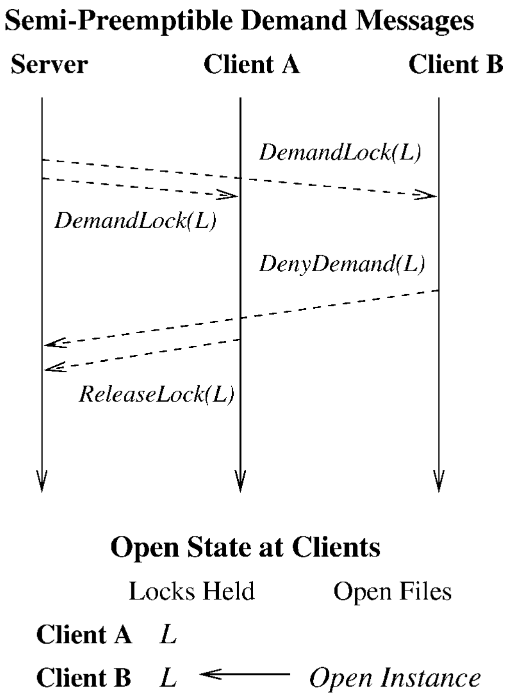

Figure 2. Demands for the semi-preemptible lock are accepted or denied depending upon client session state.

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 holds an open instance of that file, it requires the held lock to protect that instance and denies demand requests from the server (figure 2). However, if no process holds an open instance, 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 lock is held, a client can directly access the data from shared storage, and need not interact with the server at all. Without direct access storage, clients must interact with servers for data, and these file systems cannot save a message from semi-preemptible locking.

## 4. Managing complexity

Lock management includes both the locking protocol and the lock evaluation. Protocol describes how locks are obtained and released and, in our system, we use the semi-preemptible protocol. Lock evaluation is the enforcement of semantics of the locks themselves and the semantics are encoded in the lock mode. For file session locks, the mode defines the actions that are allowed under a lock and actions that are restricted by concurrent sessions. In a file system, there are a large number of possible locking modes, describing all possible combinations of actions and restrictions.

Evaluation of locks becomes difficult as the number of locking modes grows large. Techniques that reduce the number of locking modes have been explored as an alternative to implementing the full complement of possible modes. Although these techniques reduce complexity, they negatively affect concurrency and reduce performance. In Storage Tank, we implement all possible locking modes for the greatest degree of concurrency and application correctness. However, we describe other techniques for comparison.

### *4.1. Implementing subsets*

By implementing a chosen subset of all possible locking modes, some systems trade reduced complexity for minor semantic violations. File systems often select a set of distributed locks for file access that are simpler than the semantics of the underlying file open system call [7,14]. The mismatch between open semantics and locking results in either concurrent opens being allowed that violate open semantics, or concurrent opens being disallowed that open semantics would permit. File systems opt to disallow legal opens because this policy impinges on concurrency rather than correctness.

An extreme example of a subset is to protect an object with a single exclusive lock. This lock is very easy to reason about, implement, and manage. One client holds it at a time, and it gives that client total control over the locked object. However, this lock allows no concurrent action by other clients. For sessions, this would mean that there can only be one open instance of a file at a time. Single exclusive locks operate reasonably for a data consistency protocol, in which locks can be preempted. However, for non-preemptible locks, the concurrency restrictions are intolerable.

### *4.2. Multiple simple locks*

Breaking a single complex lock into multiple locks with simpler semantics is another technique for reducing complexity. While this strategy is desirable, because each individual lock has few modes and therefore is easy to manage, it introduces either deadlock or livelock and ultimately limits performance. Problems arise as the locks are not truly independent. Applications require multiple locks to be held concurrently for their operation, and application semantics tie seemingly independent locks together.

Any complex set of locking modes for a single lock can be implemented as multiple locks each with simple semantics that are obtained individually. For example, a lock that protects a data file for both writing data and updating metadata can be broken down into two separate locks, one for writing and one for metadata update. However, both locks cannot be obtained atomically.

For example, clients that need to obtain multiple preemptible locks to conduct an operation can experience livelock and have no guarantee that they will ever make any progress. Consider two clients trying to obtain the same set of locks. Both clients hold some locks currently and request the remaining locks. As each client obtains some locks, other locks are revoked. Neither client ever has the guarantee that it will get the full complement of locks needed to continue. Increases in the number of clients, resource contention, and number of locking modes all exacerbate livelock.

Techniques that eliminate livelock introduce deadlock. To eliminate livelock, clients that need multiple locks can deny or ignore revocation on its currently held locks while awaiting the other locks it requires. The same two clients in our livelock example, each requesting the same set of locks, would be at an impasse if they ignored revocation. Neither client releases the locks it holds, and neither client obtains the locks it needs. Deadlock is generally handled through either avoidance or detection [11]. Regardless of the chosen technique, dealing with deadlock always incurs a performance penalty.

Protecting all accesses using a single lock with more complex semantics, rather than multiple individual locks, avoids deadlock, livelock, and the associated inefficiencies. Rather than needing to obtain multiple locks, applications atomically obtain a single lock. The request can be evaluated immediately at the server and granted or denied. Application and lock state always progress.

## 5. Compatibility tables

For comparison against our system of dynamic evaluation, we present the compatibility table which has been developed to manage distributed concurrency in databases [11]. Our key criticism of this structure is that it does not scale well when the number of locking modes becomes large. To be fair, in databases the number of locking modes tends to be very small (in contrast to file systems) and these static data structures are more than adequate.

A compatibility table determines whether incoming lock requests can be granted in conjunction with currently outstanding locks. A system defines a set of locks and fills out this data structure, which defines how these locks are managed.

Table 1
Locking modes and the compatibility table for session locks.

| Symbol | Name | Description |
|---|---|---|
| $r$ | Read | Reader lock |
| $w$ | Write | Reader and writer lock |
| $s$ | Shared | Reader lock, no writers |
| $u$ | Update | Writer lock, no writers |
| $x$ | Exclusive | No readers or writers |

| Requested | Held | | | | | |
|---|---|---|---|---|---|---|
| | None | $r$ | $s$ | $w$ | $u$ | $x$ |
| $r$ | + | + | + | + | + | − |
| $s$ | + | + | + | − | − | − |
| $w$ | + | + | − | + | − | − |
| $u$ | + | + | − | − | − | − |
| $x$ | + | − | − | − | − | − |

When opening a file, clients select a locking mode that best matches the semantics of its open and requests the corresponding lock from a locking server. The server uses the compatibility table to evaluate whether the incoming request can be serviced. We present an example based on the locking modes and compatibility table in table 1, which are a subset of all possible locks. The table consists of rows that index the mode of the incoming lock request and columns that index locks currently held by other clients. The table cells that hold a plus indicate that the request is compatible with the outstanding lock, or a minus indicating that the requested and held lock are in conflict. For example, if a server receives a request for a $w$ (*write*) lock on a file with outstanding locks $r$ (*read*) and $s$ (*shared*), it evaluates the locks as follows. It first looks up the cell for requested mode $w$ and held mode $r$ and sees that the lock modes are compatible. It then continues to evaluate $w$ against $s$ and, seeing that they are incompatible, the $w$ lock cannot be immediately granted given current lock state. The server demands the $s$ lock and grants the $w$ lock if the demand succeeds.

For $n$ locking modes, the compatibility table contains $n^2$ entries. For databases and other systems with few locks, programming a compatibility table presents no problems. However, when we looked at implementing this structure for all possible locks in a distributed file system, the amount of state was daunting, and we were concerned with the correctness and maintenance of our implementation.

## 6. Dynamic lock evaluation

To reduce the memory used by static locking data structures, we define simple algorithms for evaluating the compatibility of lock requests. These algorithms require the client and server to store only the number of allowable access methods, $k$, and need not record every lock mode and the associated compatibility table. These algorithms support all possible locking modes for a given set of access modes, unlike systems that implement a subset of all possible locks to limit complexity.

Our algorithms use the following quantities:

| | |
|---|---|
| $\mathcal{L}$ | set of locking modes |
| $\mathcal{A}$ | set of valid access modes |
| $\forall X \in \mathcal{L}$: $P_X \subseteq \mathcal{A}$ | $X$'s permitted access modes |
| $\forall X \in L$: $D_X \subseteq \mathcal{A}$ | $X$'s disallowed sharing modes |

Each lock $X$ can be appropriately thought of as an ordered pair $\langle P_X, D_X \rangle$ of the access modes that it *permits* and the access modes it *disallows*. Permitted modes are the access methods that the lock holder can perform. Disallowed modes are the access methods that the lock holder forbids other clients from performing concurrently. Both the permitted and disallowed access modes vary over all possible subsets of the set $\mathcal{A}$. Thus for $\mathcal{A}$ containing $k$ different access modes, there are $2^{2k}$ different potential locks, because the number of distinct subsets of $\mathcal{A}$ equals $2^k$.

To develop algorithms for evaluating lock state, we begin by casting the definition of lock compatibility in set theoretic terms. From this definition, we create simple algorithms for evaluating lock requests at the server and processing local open requests and server lock demands at the client. Lock compatibility is evaluated in a dynamic fashion; i.e., compatibility need not be pre-computed and stored in tables.

**Definition 6.1 (Compatibility).** Lock $X$ and $Y$ are compatible iff $P_X \cap D_Y = \emptyset$ and $P_Y \cap D_X = \emptyset$.

Two locks are compatible if they do not forbid the access modes that the other lock protects. Compatibility must be symmetric so that the lock state in a distributed system has no dependence on the order in which locks are acquired. This helps avoid non-deterministic behavior due to race conditions. For illustration, consider that compatibility were not symmetric. We could have two locks, $X$ and $Y$, with $X$ compatible with $Y$ but not vice versa. If two clients were obtaining these locks and lock $X$ were obtained first, then another client could get lock $Y$. On the other hand, if $Y$ were obtained first, $X$ would be unavailable. The final lock state would vary depending upon the order in which the lock requests arrive.

In addition to lock compatibility, used to evaluate lock requests, clients require the concepts of lock strength and weakness to manage locks hierarchically and determine lock transitions. Strength and weakness are used by clients when either upgrading a held lock to service a local file open request or downgrading a lock in response to the server's demand.

**Definition 6.2 (Strength).** Lock $X$ is *stronger* than lock $Y$ iff $P_Y \subseteq P_X$ and $D_Y \subseteq D_X$.

**Definition 6.3 (Weakness).** Lock $X$ is *weaker* than $Y$ iff $Y$ is stronger than $X$.

A stronger lock permits more access methods and restricts sharing when compared with a weaker lock. The strength and weakness definitions include identical locks;

i.e., two identical locks are both mutually stronger than each other and weaker than each other. Although this abstraction seems incongruous, it simplifies locking algorithms by addressing boundary conditions.

The following theorems help show that these definitions match our intuition for file access locking and have semantics identical to the compatibility, strength, and weakness of static locking data structures. First, a lock hierarchy can only be valid if the strength and weakness relations are transitive, required conditions for an ordering.

**Theorem 6.1.** $X$ stronger than $Y$ and $Y$ stronger than $Z$ implies $X$ stronger than $Z$.

*Proof.* $X$ stronger than $Y$ implies that $P_Y \subseteq P_X$ and $D_Y \subseteq D_X$. Also, $Y$ stronger than $Z$ implies that $P_Z \subseteq P_Y$ and $D_Z \subseteq D_Y$. From these, we observe that $P_Z \subseteq P_X$ and $D_Z \subseteq D_X$, which is exactly the condition for $X$ stronger than $Z$. □

**Corollary 1.** $X$ weaker than $Y$ and $Y$ weaker than $Z$ implies $X$ weaker than $Z$.

Lock strength is closely related to compatibility and will be used to help determine transitions among locking modes in our system. This next theorems express key concepts that clients use to summarize locks and evaluate lock transitions.

**Theorem 6.2.** If $X$ is weaker than $Y$ then all locks compatible with $Y$ are compatible with $X$.

*Proof.* When $X$ weaker than $Y$, $P_X \subseteq P_Y$ and $D_X \subseteq D_Y$. Also, for any lock $Z$ compatible with $Y$, $P_Y \cap D_Z = \emptyset$ and $P_Z \cap D_Y = \emptyset$. This implies that $P_X \cap D_Z = \emptyset$ and $P_Z \cap D_X = \emptyset$. We conclude that $X$ is compatible with $Z$. As we have not constrained $Z$, $X$ is compatible with any lock compatible with $Y$. □

**Corollary 2.** If $X$ is not compatible with $Y$ then all locks stronger than $X$ are also incompatible.

*Proof.* For the sake of contradiction, conjecture an $Z$, stronger that $X$, that is compatible with $Y$. By theorem 6.2 $Z$ compatible with $Y$ implies that $X$ is compatible with $Y$. This establishes the contradiction and no such $Z$ can exist. □

The presented description of locks defines the semantics that must be implemented by lock evaluation algorithms. The algorithms for dynamic lock evaluation compare lock requests and transitions to these definitions, rather than performing a look up in a preset data structure. This eliminates the memory required to store lock tables.

## 7. A summarizing data structure for lock evaluation

In addition to the formal definition of lock semantics, we require data structures at the lock requester (client) and lock granter (server) to fully specify algorithms. For lock evaluation, the server has the task of receiving incoming lock requests and evaluating the compatibility of these requests against currently outstanding locks. Obvious implementations might elect a data structure that enumerates all outstanding locks and, when receiving a new lock request, the server iterates over existing locks comparing compatibility with the requested lock. Implementations based on lock tables work in exactly this fashion, performing a table look up that compares the requested lock against each currently outstanding lock.

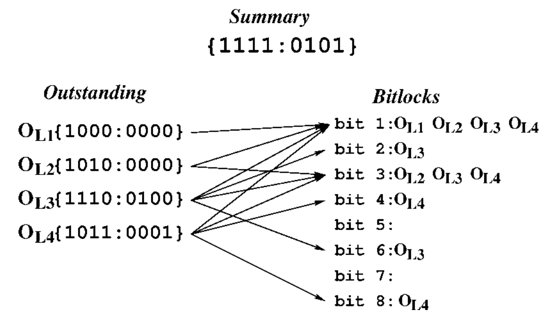

Figure 3. Lock management data structure.

We define a data structure that allows our algorithms to evaluate lock compatibility directly, without iteration. The fundamental concept of this data structure is to summarize the sharing restrictions and permitted access modes of all outstanding locks in a single location and evaluate incoming lock requests against the summary. The data structure also allows the server to find all locks that need to be demanded directly, without searching.

This concept of summarizing lock state is similar, but more general than, using the strongest outstanding lock as a representative for all outstanding locks [11]. However, our system makes provisions for locks that are compatible yet not strength-related, which occurs when all possible combinations of access modes are allowed.

In addition to the summary itself, the data structure contains a list of all outstanding locks which indexes the bits of each individual lock (figure 3). This allows the data structure to evaluate new lock requests against the summary, but also permits the contribution of any individual lock to the summary to be calculated and removed when the lock is released.

In the lock summary data structure, we use a bit vector to represent any individual lock. For a locking system with $k$ unique access modes, the bit vector contains $2k$ bits. The first $k$ bits correspond to the set of access modes that a lock permits. The next $k$ bits indicate the set of access modes that the lock disallows. For example, in a locking system with metadata read, metadata write, read, and write access modes $(m_r, m_w, r, w)$ the bit vector $\langle 1011 : 0001 \rangle$ describes a lock that permits metadata read and data read and write while disallowing other clients from holding a write privilege. We punctuate the bit vector with a colon to indicate the semantic differences between the first and second $k$ bits.

The data structure contains a list, called the Outstanding list, of all individual locks currently held by clients. The list holds a description of each lock, its $2k$ bits describing permitted access and concurrently disallowed locking modes $(\langle \vec{P}_{L_i}, \vec{D}_{L_i} \rangle)$, and is indexed by lock identifier $(L_i)$. $\vec{P}$ stands for the bit vector representation of set $P$. The server picks lock identifiers for incoming lock requests, and all subsequent lock operations are conducted by lock identifier. Since all client/server lock operations are conducted in the context of lock identifier, the server can always look up entries in this list quickly. It is possible to maintain this list as an ordered index, so that locks can be looked up by name in O(1), but engineering restrictions[2] often require unique, non-decreasing lock identifiers. These lock identifiers are maintained using extendible hashing [8] for scalability and fast look up, in time $\mathrm{O}(\log n)$ for $n$ outstanding locks.

The data structure also contains a summary of all outstanding locks. The lock summary consists of the union of all permitted access modes and the union of all disallowed concurrently held modes. For a system with locks $L_1, \ldots, L_n$ outstanding, the summary is:

$$S = \langle P_S, D_S \rangle = \left\langle \bigcup_{j=1}^{n} P_{L_j}, \bigcup_{j=1}^{n} D_{L_j} \right\rangle. \tag{1}$$

In our data structure, we represent the summary using bit vectors for sets, as if it were a single lock. The set operations for union and intersection are computed using logical operators on bit vectors: "bit-wise or" $\vee$ for union and "bit-wise and" $\wedge$ for intersection. These bit operators can be legally applied to two vectors of the same length and operate pairwise on the bits in each vector, producing a result bit vector of the same length.

Our implementation encodes the two unions of the summary in terms of bit operations, whose bit components can be expanded to arrive at an equivalent logical expression.

$$S = \langle \vec{P}_S, \vec{D}_S \rangle = \langle \vec{P}_{L_1} \vee \vec{P}_{L_2} \vee \cdots \vee \vec{P}_{L_n}, \\ \vec{D}_{L_1} \vee \vec{D}_{L_2} \vee \cdots \vee \vec{D}_{L_n} \rangle. \tag{2}$$

By maintaining the union of the permitted and disallowed lock modes at all times, the server can implicitly evaluate the compatibility of an incoming lock request against all locks by evaluating the request against the summary. A requested lock $R = \langle P_R, D_R \rangle$ is compatible if it does not disallow any access modes permitted by the summary, and the summary does not disallow any permitted modes it accesses, i.e.,

$$(P_R \cap D_S = \emptyset) \wedge (D_R \cap P_S = \emptyset)$$

which is equivalent to

$$(\vec{P}_R \wedge \vec{D}_S) \vee (\vec{P}_S \wedge \vec{D}_R) = \vec{0}\ . \tag{3}$$

The intuition behind the summary is that all the lock state in the system can be represented by an equivalent single lock, constructed from the union of all permitted and concurrently prohibited access modes.

In addition to the summary and Outstanding lists, we require additional lists to aid in processing lock requests and efficiently maintaining the summary. For each bit in the lock summary, we keep a BitLocks structure that contains a list of the locks in Outstanding that set that bit high. Each BitLocks list describes the set of locks that contribute to setting an individual bit of the summary high. Our algorithms for lock management use this list to efficiently determine what locks need to be revoked when a requested lock is not compatible with current lock state.

Each lock in Outstanding also has references (pointers) to the locations in all BitLocks lists in which the lock appears. When a lock is released, the lock management algorithms use these references to quickly remove the released lock from all BitLocks lists. The references actually point from the entry in the Outstanding list to the actual entry in the BitLocks list and not to the head of a list as the diagram might indicate. This allows the algorithm to look up an entry in the BitLocks list in unit time.

### *7.1. Correctness*

To establish that evaluating lock requests based on this summary (equation (3)) is correct, we show that the summary expression can be derived as a logical simplification of the compatibility criteria: a requested lock is compatible with outstanding lock state and can be granted if and only if it is compatible (definition 6.1) with all outstanding locks.

For a system with locks $L_1, \ldots, L_n$ outstanding, the requested lock must be evaluated for compatibility pairwise against locks $L_1, \ldots, L_n$.

$$\forall L_j \in \{L_1, \ldots, L_n\}: \\ (P_R \cap D_{L_j} = \emptyset) \wedge (D_R \cap P_{L_j} = \emptyset) \tag{4}$$

$$\equiv \bigwedge_{j=1}^{n} \left( (\vec{P}_R \wedge \vec{D}_{L_j} = \vec{0}) \wedge (\vec{P}_{L_j} \wedge \vec{D}_R = \vec{0}) \right) \tag{5}$$

$$\equiv \bigvee_{j=1}^{n} \left( (\vec{P}_R \wedge \vec{D}_{L_j}) \vee (\vec{P}_{L_j} \wedge \vec{D}_R) \right) = \vec{0} \tag{6}$$

$$\equiv \left( \vec{P}_R \wedge \bigvee_{j=1}^{n} \vec{D}_{L_j} \right) \vee \left( \bigvee_{j=1}^{n} \vec{P}_{L_j} \wedge \vec{D}_R \right) = \vec{0} \tag{7}$$

$$\equiv (\vec{P}_R \wedge \vec{D}_S) \vee (\vec{P}_S \wedge \vec{D}_R) = \vec{0}\ . \tag{8}$$

Noting that (8) is equivalent to (3), this transformation shows that evaluation against the summary is equivalent to evaluation against all outstanding locks and is the original motivation for the summarizing data structure. We first developed the formalism for locking presented in section 6. Upon seeing the expression for evaluating the compatibility of a single (requested) lock against a set of compatible (outstanding) locks, we determined that it can be simplified. The summarizing data structure for lock evaluation merely captures the logical simplification.

## 8. Algorithms for lock management

Based on both dynamic lock evaluation and the semi-preemptible protocol, we develop a set of algorithms for managing a distributed set of locks. These algorithms are based on the summarizing data structure and logical rules for lock evaluation. They also capture our design goal of eliminating lock state using hierarchical locking.

In Storage Tank's semi-preemptible protocol, we restrict each client's lock holdings to a single distributed session lock for each file. Clients may have many local open instances of that given file protected by a single lock.

Clients 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. 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 concepts of strength and weakness help the system determine what action to take, i.e., how to change lock state to service local open requests or server demand messages.

We note that upgrade does not lead to deadlock for file session locks. In databases, the term upgrade is sometimes used interchangeably with the term *promotion* [11], which describes a process reading a data object under one lock and promoting the lock before writing. This type of promotion leads to deadlock. For example, if two processes reading the same object both hold the weaker lock and require the stronger lock for writing, then neither will receive the promoted lock, because neither will release the weaker lock which protects the contents of the object that it has read. In contrast, for file session locks an upgrade changes the abstract distributed lock and does not modify the mode of an actual session; i.e., upgrades are not promotions. Conflicting upgrades of session locks do not result in deadlock, because the semi-preemptible protocol allows the server to deny conflicting lock requests.

```
EvaluateLockRequest
Inputs: The requested lock $R = \langle \overrightarrow{P_R}, \overrightarrow{D_R} \rangle$ and the lock summary data structure.

1. If $(\overrightarrow{P_R} \wedge \overrightarrow{D_S}) \vee (\overrightarrow{P_S} \wedge \overrightarrow{D_R}) = \overrightarrow{0}$, the lock is compatible. Proceed to step 5 to grant the lock.
2. Else Demand all incompatible locks.
    (a) For all non-zero bits in $\overrightarrow{P_R} \wedge \overrightarrow{D_S}$ demand the locks from the summary that disallow the access modes desired by the requested locks, i.e., for $i = 1, \ldots, k$: if $P_R(i) \wedge D_S(i)$ then demand all locks in BitLocks($i + k$).
    (b) Similarly, for all non-zero bits in $\overrightarrow{P_S} \wedge \overrightarrow{D_R}$ demand the locks from the summary that permit access for the bit the requested lock disallows, i.e., for $i = 1, \ldots, k$: if $P_S(i) \wedge D_R(i)$ then demand all locks in BitLocks($i$).
3. Receive and process all responses to demand requests using function ReleaseLock or by downgrading the lock.
4. If any lock holder refuses the demand, deny the lock request and Return.
5. Grant the requested lock. Reflect the change in lock state on the summary data structure using GrantLock and Return.
```

Figure 4. Server routine for evaluating incoming lock requests.

### *8.1. Server algorithm*

The server acts as a central management authority for granting and revoking locks. It takes incoming lock requests from clients, takes action by revoking outstanding locks to make the requested lock compatible, and grants the request when possible. To perform these actions, the server uses the summarizing locking data structure.

The server receives lock requests from multiple clients concurrently and processes these requests serially. Distributed locking requires that all lock state changes occur atomically. For this reason, the server considers one lock operation at a time. Multiple requests for the same lock are queued at the server and executed serially. Processing one lock request may result in multiple changes in lock state. That is, before granting a lock a server may demand multiple locks and reflect many state changes on the summary. However, these are server-driven state changes rather than client requests and, therefore, are conducted in the context of a single client-initiated lock request.

Upon receiving an incoming lock request, the server must evaluate the requested lock for compatibility and demand incompatible locks before granting the request. The server follows the steps in figure 4 to determine the compatibility of the incoming lock request with respect to current system state. The server first evaluates the lock against the summary (step 1) to determine if the lock is compatible with all outstanding locks and can be granted immediately.

If the request cannot be immediately fulfilled, one or more currently outstanding locks conflict with the request and must be demanded. The server determines and demands the set of conflicting locks in step 2. Outstanding locks can be incompatible because they disallow one of the access modes required by the requested lock (step 2(a)). Alternatively, outstanding locks can be incompatible because

**GrantLock**

*Inputs:* The lock to be granted $G = \langle \overrightarrow{P_G}, \overrightarrow{D_G} \rangle$ and the lock summary data structure.

1. Register the lock to be granted in the list of outstanding locks. Create entry $O_G = \langle \overrightarrow{P_G}, \overrightarrow{D_G} \rangle$ in the list Outstanding.
2. Calculate the contribution of the lock to be granted to the summary.
   (a) $\overrightarrow{P_S} \leftarrow \overrightarrow{P_S} \vee \overrightarrow{P_G}$.
   (b) $\overrightarrow{D_S} \leftarrow \overrightarrow{D_S} \vee \overrightarrow{D_G}$.
3. Add a reference to $G$ in the BitLocks data structures for each bit it permits and each it disallows. For each entry add a link from $O_G$ to the BitLocks list.
   (a) For $i = 1, \ldots, k$ : if $D_G(i) = 1$ add $O_G$ to BitLocks$(i + k)$.
   (b) For $i = 1, \ldots, k$ : if $P_G(i) = 1$ add $O_G$ to BitLocks$(i)$.

Figure 5. Server routine for updating the summarizing data structure when granting a lock.

**ReleaseLock**

*Inputs:* The lock to be released $R = \langle \overrightarrow{P_R}, \overrightarrow{D_R} \rangle$ and the lock summary data structure.

1. Look at the references in $O_R$ to find and remove all occurences of $O_R$ in the BitLocks lists.
2. If any of the BitLocks lists become empty, set the corresponding bits low in the summary, i.e., if BitLocks$(i) = \emptyset$ then $S(i) \leftarrow 0$.
3. Remove $O_R$ from the Outstanding list.

Figure 6. Server routine for processing released locks.

| Server | | Client 1 | | Client 2 | | Client 3 | |
|---|---|---|---|---|---|---|---|
| $S$ | Summary | $H_1$ | Held | $H_2$ | Held | $H_3$ | Held |
| $H_1$ | Client 1 | $S_1$ | Summary | $S_2$ | Summary | $S_3$ | Summary |
| $H_2$ | Client 2 | $L_{1,1}$ | Local | $L_{2,1}$ | Local | $L_{3,1}$ | Local |
| $H_3$ | Client 3 | $L_{1,2}$ | Local | $L_{2,2}$ | Local | $L_{3,2}$ | Local |
| | | $L_{1,3}$ | Local | $L_{2,3}$ | Local | | |
| | | $L_{2,3}$ | Local | | | | |

$S = H_1 \vee H_2 \vee H_3$
$S_1 = L_{1,1} \vee L_{1,2} \vee L_{1,3}$
$S_2 = L_{2,1} \vee L_{2,2} \vee L_{2,3} \vee L_{2,4}$
$S_3 = L_{3,1} \vee L_{3,2}$

Figure 7. Hierarchical lock management among client and servers.

they permit an access mode the requested lock disallows (step 2(b)). The routine evaluates the conflicting access modes using the requested lock and the summary. For all high bits in $\overrightarrow{P_R} \wedge \overrightarrow{D_S}$ and $\overrightarrow{P_S} \wedge \overrightarrow{D_R}$, the routine demands all locks that set those bits. The remainder of the routine implements the semi-preemptible lock protocol through client/server interactions. If any client denies a demand, the requested lock is refused. Otherwise, if all demands succeed, the server grants the lock request (step 5).

The routine to evaluate lock requests results in outstanding locks being released and a new lock being granted, which changes the global lock state. The server updates the locking data structure to reflect these changes using subroutines when granting a lock (figure 5) and when clients release locks (figure 6).

When granting a lock, the server updates the locking data structures to reflect the new outstanding lock (figure 5). The new lock is added to the list of outstanding locks (step 1). The summary is modified to reflect the change in lock state (step 2). Finally, for every access mode permitted or disallowed, the new lock must be added to the BitLocks list and references added from the lock entry in Outstanding to the entries in BitLocks.

When clients release locks, the server updates its data structure using the routine in figure 6. Releasing a lock consists of removing the lock from the list of Outstanding locks (step 3), updating the summary to reflect the change of state (step 2), and removing references to the lock (step 1).

We do not present algorithms for processing upgrade requests and downgrade requests because these operations can be treated as combinations of release and request. The important difference between upgrade (or downgrade) and a combination of simpler operations is that the upgrade (or downgrade) must be performed atomically. Atomicity for multiple operations can be achieved at the server by allowing only a single process to access the locking data structure at a time. However, unlike the server algorithms, the client/server protocol must contain upgrade and downgrade primitives, as the protocol has no atomicity guarantee across multiple operations.

### 8.2. Client algorithms

The semi-preemptible locking protocol allows clients to hold locks even when they do not use them. By holding locks when not in use, caching locks, the client optimistically retains privileges on the premise that the last client to lock a resource is the most likely to lock it again. This design aims to minimize changes in lock state and the associated messaging and transactional overhead.

The client uses two techniques to manage distributed locks and local processes that operate under these locks. First, the client allows local processes to use any subset of the access modes allowed by the locked resource; i.e., the client is not required to use the full strength of the held lock. Also, the client allows multiple local processes to operate under the same distributed lock.

To implement these techniques, the client manages its held lock and local processes hierarchically. The client holds a distributed lock $(H)$ with a server (figure 7). $H$ encapsulates all actions that local processes can take against the locked resource. The local processes only access data using the modes specified by the lock and only prohibit the concurrent access modes encoded in $H$. Under this held lock, the client maps the access modes and sharing limitations of each local process to a *local lock*. The local lock is a local management abstraction and is not visible to other clients or the server. A client holds many local locks and uses the same summarizing data structure as the server (figure 3) to evaluate and manage local locks. For the client, the summary describes all in-use access modes and disallowed modes for local processes accessing the data. The summary is restricted to be always weaker than the distributed held lock. This allows the held lock to protect concurrency guarantees expressed in the summary and, therefore, needed by the local locks.

**ProcessDemand**

*Inputs:* The remotely requested lock included in the demand message $R = \langle \overrightarrow{P_R}, \overrightarrow{D_R} \rangle$ and the client lock summary data structure.

1. Evaluate whether the requested lock is compatible with the currently held local locks. If $(\overrightarrow{P_R} \wedge \overrightarrow{D_S}) \vee (\overrightarrow{P_S} \wedge \overrightarrow{D_R}) = \vec{0}$ proceed to step 2, Else deny the demand and Return.
2. Determine the downgraded lock mode required to protect the currently held locks and the requested lock using one of the two following heuristics.
   - (a) Maximum downgrade heuristic, $H = \langle \overrightarrow{P_S}, \overrightarrow{D_S} \rangle$.
   - (b) Minimum downgrade heuristic, $H = \langle \overrightarrow{P_H} \wedge \neg \overrightarrow{D_R}, \overrightarrow{D_H} \wedge \neg \overrightarrow{P_R} \rangle$
3. Inform the server of the downgrade and Return.

Figure 8. Client routine to determine the needed lock downgrade.

The management of the client's summarizing data structure is different than the server's. In many ways, management is simpler because the local locks held by local processes are non-preemptible. This eliminates demands or revocations of held locks. In other ways, management is more complex because when the client changes local lock state, it must consider the distributed system state, encapsulated by the held distributed lock $H$.

The main principle in the management of multiple local locks under a single distributed lock is theorem 6.2, which states that if a lock is compatible with all other outstanding locks in the distributed system then all locks weaker than that lock are also compatible. This ensures that if the remainder of the distributed system acts in accord with the held lock, the permitted access and sharing of the local locks are respected.

Locking clients respond to several stimuli, requests to obtain and release local locks from local processes, and demand requests from the lock server. Clients obtain and upgrade distributed locks on behalf of local processes that require stronger local locks and downgrade and release locks in response to server demands. In effect, the client acts as a server of locks to local processes, under the constraint that the client holds a suitable distributed lock. For this reason, lock acquisition, release, upgrade, and grant operations at the client use routines much like the server's. Owing to similarities, we omit detailed descriptions of these operations.

The remaining client action is to process demand requests from the server (figure 8). Demands are initiated by a server attempting to alter the distributed lock state in order to grant a lock to another client. The lock mode requested by the other client is included in the server's demand message so that the requested lock can be evaluated against current open state. In step 1, the client evaluates whether the requested lock can be granted given local lock state. Only the clients know local state and they must evaluate the request and determine the conflicting outstanding locks. The clients themselves make the final determination as to the compatibility of the request against current system state by comparing the requested lock against their current local lock state.

Clients with compatible local locks downgrade their distributed lock. When choosing the locking mode for the downgrade, there are many possible choices. We present the two extreme choices as possible heuristics. First, a client can elect to retain as strong a lock as possible, the minimum downgrade heuristic. The client reduces its holding as little as possible, in the lock strength sense, to make its lock compatible with the requested lock. This choice would perform well if future access to the object were more likely to occur locally than from other clients. However, if the interest in the object has moved to other clients, the policy may result in future demands. The other alternative is to downgrade the lock holding as much as possible: the maximum downgrade heuristic. The downgraded locking mode is exactly the lock summary – the fewest permitted access modes and disallowed modes needed by local locks.

## 9. Comparing dynamic evaluation with traditional locking

We present the execution time and memory usage bounds of lock evaluation algorithms to illustrate how dynamic evaluation asymptotically outperforms evaluation based on static data structures if locking systems are complex.

The amount of space used by a locking system can be divided into static and dynamically used space. The static space contains the information the system needs to know about locking modes to properly run the locking protocol. The dynamic space contains information about current outstanding locks. Conventional wisdom holds that the dynamic costs dominate the static costs. However, as locking systems become more complex, this assertion is not always true because the static space costs can grow exponentially in the number of access modes.

For static data structures, $k$ different access modes results in $2^{2k}$ unique locks. Locking with traditional data structures requires space in $\Omega(2^{4k})$. There are $2^{4k}$ entries in a lock compatibility table for $2^{2k}$ locks. For dynamic lock evaluation, the algorithms only need to know the different locking modes, which uses space in $\mathrm{O}(k)$.

The dynamic space usage of locking using traditional data structures is $\Theta(nk)$ for $n$ outstanding locks. The system stores a list of the $k$ modes of the $n$ outstanding locks. Dynamic evaluation uses the summarizing data structure to hold dynamic lock state, which is an $\Theta(nk)$ data structure. The Outstanding list has $n$ entries each of size $k$, and there are $k$ different BitLocks lists with a total of $n$ entries in all lists and $n$ incoming pointers referring to these entries.

For execution time, we look at how long operations take with respect to the amount of outstanding lock state $n$ and the amount of static state $k$. Lock evaluation with static data structures takes time in $\Omega(n)$, because the request must be checked against all outstanding locks. Dynamic lock evaluation uses execution time in $\mathrm{O}(k)$ to evaluate the lock, update the lock summary, and fill out the BitLocks and Outstanding lists. Similarly, releasing a lock takes time in $\mathrm{O}(k)$. When concurrency is high, there are many outstanding locks, $n$ dominates $k$, and dynamic evaluation operates more efficiently.

The total space required for dynamic evaluation of locks is always asymptotically less than or equal to that required for evaluation based on static data structures. However, the dynamic space requirements of dynamic evaluation of locks are larger by a constant factor. The list of locks used for static evaluation is equivalent in size and content to the Outstanding list. All other portions of the summarizing data structure are overhead. The true space savings are realized on the static data structures and become important when $k$ grows large.

Perhaps more important than the space and time savings is the improvement in manageability and ease of implementation. An implementer using a lock compatibility table must reason about each pair of locks and complete the entry in the table. This task can be difficult even for as few as 64 locks in Windows NT. Additionally, all locking modes must be preprogrammed. With dynamic evaluation, the implementer need only identify how the object will be accessed. The dynamic evaluation algorithms manage concurrency and compatibility.

## 10. Related research

Many distributed file systems clients transact with a server on every open and close of a file for synchronization [5,24, 25]. 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 [7] 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.

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 [17,20,23], Storage Tank uses a lease-based [6] protocol to ensure operational safety and high availability in the presence of client and server failures, and network partitions.

## 11. 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. Semi-preemptible locks are also 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.

We have also presented a set of algorithms and data structures for the dynamic evaluation of locks. Dynamic evaluation provides a formal and compact framework for evaluating lock compatibility at runtime. Dynamic evaluation obviates lock compatibility tables, exponentially reducing the space requirements to describe a locking system. It also replaces ad hoc systems for lock evaluation, providing a formalism that manages locks correctly and without deadlock.

In combining semi-preemptible locks and dynamic evaluation, we present a total solution for managing sessions in a distributed file system. Our techniques are space efficient, come with correctness guarantees, and exploit file system workload characteristics.

## Notes

1. We use the term “demand” for consistency with existing 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.
2. Security [19,21] and failure recovery [3] benefit from unique and nondecreasing lock identifiers.

## 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) (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: *Proc. of the 13th Annual Symposium on Operating Systems* (October 1991).

[3] M.L.G. Baker, Fast Crash Recovery in Distributed File Systems, Ph.D. thesis, University of California at Berkeley (1994).

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

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

[6] R.C. Burns, R.M. Rees and D.D.E. Long, An analytical study of opportunistic lease renewal, in: *Proc of the 16th Int. Conf. on Distributed Computing Systems* (2001).

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

[8] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, Extendible hashing: A fast access method for dynamic files, ACM Transactions on Database Systems 4(3) (1979).

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

[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] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques* (Morgan Kaufmann, San Mateo, CA, 1993).

[12] B. Gronvall, A. Westerlund and S. Pink, The design of a multicast-based distributed file system, in: *Proc. of the 3rd Symp. 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) (February 1988).

[14] M.L. Kazar, Synchronization and caching issues in the Andrew file system, in: *Proc. 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: *Proc. 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] 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).

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

[19] C. Neumann and T. Ts'o, Kerberos: An authentication service for computer networks, IEEE Communications Magazine (September 1994).

[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: *Proc. of the 16th IEEE Mass Storage Systems Symposium* (1999).

[21] B.C. Reed, D.D.E. Long, E.G. Chron and R.C. Burns, Authenticating network attached storage, IEEE MICRO 20(1) (January 2000).

[22] 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).

[23] C.A. Thekkath, T. Mann and E.K. Lee, Frangipani: A scalable distributed file system, in: *Proc. of the 16th ACM Symposium on Operating System Principles* (1997).

[24] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, The LOCUS distributed operating system, in: *Proc. of the 9th ACM Symp. on Operating Systems Principles* (1983).

[25] 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: *Proc. of the 1985 Winter Usenix Technical Conference* (January 1985).

[26] J. Yin, L. Alvisi, M. Dahlin and C. Lin, Volume leases for consistency in large-scale systems, IEEE Transactions on Knowledge and Data Engineering 11(4) (July/August 1999).

**Randal C. Burns** is a research staff member at the IBM Almaden Research Center. He earned a B.S. in geophysics from Stanford in 1993 and an M.S. and Ph.D. in computer science from the University of California, Santa Cruz in 1997 and 2000, respectively. Randal pursues research in distributed operating systems with a focus on storage systems, concurrency control, databases, data placement, and allocation. He is a member of the ACM and IEEE.

**Bob M. Rees** is a Senior Member of the Technical Staff at the IBM Almaden Research center. He has been building storage systems for over 20 years. He is currently the project leader for the Storage Tank project at IBM and was formerly the chief architect of the IBM Adstar Distributed Storage manager product. He earned a B.S. in computer science from the University of California, Santa Cruz in 1992.

**Larry J. Stockmeyer** received a Ph.D. in Computer Science from M.I.T. in 1974. He is currently a research staff member at IBM's Almaden Research Center in San Jose, California. Between 1974 and 1982, he was at IBM's Thomas J. Watson Research Center. He is an ACM Fellow. His research interests include computational complexity theory and storage systems.

**Darrell D.E. Long** is Professor of computer science, Associate Dean and Director of the Storage Systems Research Center at the Jack Baskin School of Engineering, at the University of California, Santa Cruz. He is also a consultant to IBM Research, where he is part of the Storage Tank research effort. He earned his M.S. and Ph.D. degrees in Computer Science and Engineering at the University of California, San Diego, and his B.S. in computer science from San Diego State University. He is the author of more than 75 research publications, and holds 4 patents.