Minuet: Rethinking Concurrency Control in Storage Area Networks

Andrey Ermolinskiy
Daekyeong Moon
Byung-Gon Chun
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-57
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-57.html

May 19, 2008

Copyright © 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Minuet: Rethinking Concurrency Control in Storage Area Networks

Andrey Ermolinskiy†, Daekyeong Moon†, Byung-Gon Chun⋆, Scott Shenker†⋆
†University of California at Berkeley, ⋆International Computer Science Institute

Abstract

Clustered applications in storage area networks (SANs), widely adopted in enterprise datacenters, have traditionally relied on distributed locking protocols to coordinate concurrent access to shared storage devices. In this report, we examine the semantics of traditional lock services for SAN environments and ask whether they are sufficient to guarantee data safety at the application level. We argue that a traditional lock service design that enforces strict *mutual exclusion* and a *globally-consistent view of locking state* is neither strictly necessary nor sufficient for ensuring application-level correctness in the presence of asynchrony and failures. We also argue that in some cases, strongly-consistent locking imposes an additional and unnecessary constraint on application availability. Armed with these observations, we develop a set of novel concurrency control and recovery protocols for clustered SAN applications that achieve safety and liveness in the face of arbitrary asynchrony, process failures, and network partitions. Finally, we present and evaluate Minuet, a new synchronization primitive based on these protocols that can serve as a foundational building block for safe and highly available SAN applications.

1 Introduction

In recent years, storage area networks (SANs) have been gaining widespread adoption in enterprise datacenters and are proving effective in supporting a range of applications across a broad spectrum of industries. Some of the common applications include online transaction processing in finance and e-commerce, digital media production, business data analytics, and high-performance scientific computing. A number of hardware and software vendors, including companies such as EMC, HP, IBM, and NetApp, offer SAN-oriented products and services to their customers.

In a SAN storage architecture, a pool of storage devices, typically disk arrays or specialized storage appliances, are exposed to a group of server nodes for shared access over a switched network. To applications running on these nodes, shared disks appear as locally-attached devices while in actuality, application's I/O requests are sent over the network to the corresponding target device using a specialized network protocol such as FibreChannel or iSCSI.

For parallel clustered applications that demand high-speed concurrent access to large volumes of data, SAN offers an attractive architecture for a scalable storage backend. In such environments, a clustered middleware service is commonly deployed on application nodes to provide a higher-level primitive such as a filesystem (GFS [1], OCFS [2], PanFS [3], GPFS [4], Lustre [5], Xsan [6]) or a relational database (Oracle RAC [7]) on top of raw disk blocks.

One of the primary design challenges in clustered SAN environments is ensuring safe and efficient coordination of access to application state and metadata that resides on shared storage. The traditional approach to concurrency control in shared-disk clusters involves the use of a synchronization module called a *distributed lock manager* (DLM). To obtain exclusive access to a particular shared resource on disk (e.g., a file, a record, or a piece of application metadata) a process must first acquire a lock on the respective resource. The DLM service provides the guarantee of mutual exclusion, ensuring that no two processes in the system are concurrent holders of conflicting locks.

In abstract terms, providing such guarantees requires enforcing a globally-consistent view of locking state and one could argue that a traditional DLM design views such consistency as *an end in itself* rather than a means to achieving application-level correctness.

In this paper, we take a closer look at the semantics of a traditional lock service for SAN clusters and ask whether the assurances of full mutual exclusion and strongly-consistent locking are, in fact, a prerequisite for correct application behavior. Our main hypothesis is that the standard semantics of mutual exclusion provided by a DLM are neither sufficient nor strictly necessary to guarantee safe coordination of access to shared state on disk in the presence of failures and asynchrony.

We propose and evaluate a new technique for disk access coordination in SAN environments. We augment target disk devices with a tiny piece of application-independent logic, called a *guard*, that rejects inconsistent I/O requests and enables us to provide a property called *session serializability*. We argue that while this correctness condition is more permissive than strict mutual exclusion, it is just as useful from the practical standpoint and corresponds to application developers' expectations. The guard logic can be used to make existing SAN protocols safe in the presence of asynchrony and process fail-

ures.

In addition, the guarantee of session serializability allows us to develop novel concurrency control and recovery protocols, which operate safely while offering the following benefits over traditional mechanisms that rely on a strongly-consistent DLM:

1. Improved availability with replicated lock managers, ensuring progress with less than a majority of replicas.
2. Reduced failure recovery times.
3. Control over the tradeoff between strong coordination and optimistic concurrency.

Finally, we describe the implementation of Minuet, a novel synchronization primitive for SAN environments based on the presented protocols. This system assumes the presence of guard logic at storage devices and provides applications with locking and transaction recovery facilities, while ensuring data safety and liveness in the face of arbitrary asynchrony, node failures, and network partition scenarios. Our evaluation shows that applications built atop Minuet compare favorably to those that rely on a conventional strongly-consistent DLM design, offering improved availability and competitive performance. We hope to demonstrate that Minuet is a useful general-purpose building block for clustered SAN applications and infrastructure components such as filesystems and databases.

The rest of this paper is organized as follows. In Section 2, we describe the relevant background on SAN and provide several examples of safety problems. In Section 3, we present our main contribution, the design of a novel safe and highly available synchronization mechanism. Section 4 describes our prototype implementation of Minuet and several sample client applications. We evaluate our system in Section 5 and and discuss practical aspects of our approach in Section 6. Finally, we provide an overview of related work in Section 7 and conclude in Section 8.

2 Background

2.1 Storage area networks (SANs)

Storage area networks are becoming increasingly popular in enterprise datacenters and are commonly adopted to support the storage needs of data-intensive clustered applications that require high-speed parallel access to shared persistent state. In the SAN (or *shared disk*) model, persistent storage devices, typically disk drive arrays or specialized hardware appliances, are attached to a dedicated *storage network* and appear to members of the application cluster as locally-attached disks. The goal is to provide fully-decentralized access to shared application state and in principle, any application node can access and issue *Read*/*Write* requests on any piece of data without routing these requests to a dedicated server. While in the shared-disk model, all I/O requests on a particular data object are centrally serialized, the crucial distinction from the *shared-nothing* paradigm is that the point of serialization is a hardware disk controller that exposes a well-defined application-independent interface on raw physical blocks and is oblivious to application semantics and data layout considerations.

Generally speaking, the shared-disk paradigm can be seen as advantageous from the standpoint of availability because it offers better redundancy and decouples processor failures from loss of persistent state. Incoming application requests can be routed to any available node in the application cluster and in the event of a node failure, subsequent requests can be redirected to the next available processor with minimal interruption of service and no long-term impact on data availability. In contrast, a server failure in the shared-nothing model may render some portions of the dataset temporarily or permanently unavailable.

One of primary challenges in designing SAN-oriented clustered applications and middleware is ensuring safe and efficient coordination of access to data that resides on shared disks and preserving correct ordering of concurrent requests from multiple processes. Commonly, a software module called a *distributed lock manager* (DLM) is employed to provide such coordination. A typical DLM service exposes a generalized notion of a *resource* - an abstract application-level entity to which access must be controlled and the goal is to guarantee that no two processes simultaneously possess conflicting locks on the same resource - a form of *group mutual exclusion* [8]. In its simplest form, the shared-exclusive locking protocol allows a group of readers and writers to coordinate their disk requests to a piece of shared data that represents some resource *R*, ensuring that every process sees a consistent image of *R*. The protocol requires a process to acquire a shared lock before issuing a *Read* request to disk and, similarly, *Write* requests must be delivered under the protection of an exclusive lock.

2.2 Safety and liveness limitations in SAN environments

In principle, a DLM service provides sufficient mechanism to ensure safe access to application state on disk, provided that every client process obeys the basic locking protocol and submits its I/O requests only when holding an appropriate lock. In practice, however, guaranteeing safe ordering of I/O requests at shared disks tends to be more difficult than the above discussion might suggest due to *process failures* and *effects of asynchrony*.

Ensuring progress in the face of process failures

requires employing mechanisms such as leases and heartbeat-based failure detection. Upon suspicion of failure, the lock manager must reclaim locks previously held by the suspected process and make them available to other clients, but inconsistent failure observations can result in prematurely-reclaimed locks and ultimately threaten data safety. Next, we provide several concrete examples that demonstrate how data corruption can arise in a failure-prone shared disk cluster.

Scenario 1: Consider a space allocation mechanism that employs a bitmap to keep track of free space on disk. An application process that needs to allocate a free datablock must read a portion of the bitmap from disk, find a zero-valued bit representing a free block, flip its value, and write the updated segment back to disk. This simple read-modify-write operation is performed under protection of an exclusive lock to ensure that no concurrent allocation attempts would select the same bit. Suppose a client process c_1 is holding a lock on a particular bitmap fragment and has chosen to flip the bit at position x in block B. c_1 is asserting its liveness status to the lock manager via a heartbeat mechanism, but suppose that because of a transient network problem between c_1 and the rest of the cluster, some of its heartbeats fail to reach the recipient in a timely manner, thereby triggering a failure suspicion event. The lock manager reacts by reclaiming the exclusive lock and granting it to c_2, which proceeds to reading B from disk. c_1's update may not have reached the disk by the time c_2's *Read* operation arrives and as a result, both nodes might select bit x, which would result in two conflicting allocations of the respective datablock. Note that in this scenario, both clients obey the basic locking protocol and the loss of data integrity could be linked to the impossibility of reliable failure detection in an asynchronous distributed setting [9].

Scenario 2: Consider two clients, c_1 and c_2, that are concurrently accessing a data structure S residing on a shared disk D in a contiguous array of blocks numbered [0-9]. Suppose c_1 is updating S under the protection of an exclusive lock and c_2 wants to prefetch the contents of S into a local memory buffer and is waiting for a shared lock on S. c_1 submits $Write(target = D, offset = 3, length = 5)$ but crashes before hearing a response and the lock manager *correctly* detects the failure and reacts by reclaiming the exclusive lock on S and granting it in shared mode to c_2. That client proceeds to reading the object and submits $Read(target = D, offset = 0, length = 5)$, which returns old data. Next, c_1's delayed $Write$ request hits the disk and overwrites data at offsets $[3-7]$, after which c_2 issues $Read(target = D, offset = 5, length = 5)$. Note that although each individual I/O request is processed as an atomic operation by the storage device, c_2 in the above scenario would observe and potentially act upon a partial $Write$ from c_1, which may be viewed as a violation of application safety.

Scenario 3: Commonly, clustered applications and middleware services need to enforce transactional semantics on updates to application state and metadata. In a shared-disk clustered environment, distributed transactions have traditionally been supported via the use of two-phase locking in conjunction with a distributed write-ahead logging (WAL) protocol and we refer the reader to D-ARIES [10] for a detailed exposition of transaction recovery in the context of a shared-disk parallel RDBMS. In the abstract, the system maintains a snapshot of application state along with a set of per-client append-only logs (also on shared disks) that record Redo and/or Undo information for all updates performed by the respective client and the commit status of every transaction. During failure recovery, the system must examine the suspected client's log and restore consistency by rolling back all uncommitted updates and replaying all updates associated with committed transactions that may not have been synced to the snapshot prior to the failure. An essential underlying assumption in a WAL-based recovery scheme is that once a failure suspicion event is delivered and the decision to initiate log recovery is made, no additional $Write$ requests from the suspected process will hit the snapshot or the log and data corruption may occur if this assumption is violated.

Ensuring application safety in a shared-disk environment has traditionally required introducing a set of synchrony assumptions, such as bounded clock drift rates and message propagation delays, that permit construction of reliable heartbeat-driven failure detectors and effectively transform an error-prone asynchronous environment into a partially synchronous one. Fundamentally, these assumptions are probabilistic at best and since application data integrity is predicated on the validity of these assumptions, failure timeouts are typically tuned to a very conservative value in order to minimize the probability of safety violation. Such (necessarily) pessimistic method of tuning timeouts may have a profoundly negative impact on failure recovery times - one of the common criticisms of SAN-oriented applications.

Another limitation commonly exhibited by DLM-supported SAN applications is *liveness*. The lock manager represents an additional point of failure and while various fault tolerance techniques can be applied to improve its availability, the very nature of the semantics enforced by the DLM places a fundamental constraint on the overall system availability. For instance, multiple lock manager replicas can be deployed in a cluster, but mutual exclusion can be guaranteed only if clients' requests are presented to them in the same order, which necessitates mechanisms such as state machine replication [11] and Paxos [12] for request ordering agreement. Alternatively, a single lock manager instance may be elected dynami-

cally [13–15] from a group of candidates and in this case, ensuring mutual exclusion requires global agreement on lock manager's identity. In both cases, reaching agreement fundamentally requires access to an active primary component - typically a majority of nodes. As as a result, a large-scale node failure or a network partition that renders the primary component unavailable or unreachable may bring about a cluster-wide outage and complete loss of service.

To summarize, today's SAN applications and middleware face significant limitations along the dimensions of safety and liveness. At present, several hardware-assisted techniques, such as out-of-band power management (STOMITH [1]) [16,17], SAN fabric fencing [18], and SCSI persistent reserve [19], can be employed to mitigate some of these issues. These mechanisms help reduce the likelihood of data corruption under typical failure scenarios, but do not provide the desired assurances of safety in the general case and, as we would argue, do not address the underlying problem. We conjecture that the underlying problem is a case of *capability mismatch* between "intelligent" application clients that possess full knowledge of application's data structures, physical disk layout, and consistency semantics on the one hand and relatively "dumb" storage devices on the other. The safety problems illustrated above can be attributed to disk controller's inability to identify and appropriately react to the various application-level events such as *lock release*, *failure suspicion*, and *failure recovery action*.

We suggest that despite this intelligence gap, the safety and liveness limitations exhibited by SAN applications today are not an inevitable property of the shared-disk architecture. Our main goal is demonstrating the feasibility of a shared disk application that ensures data safety and progress in the face of arbitrary asynchrony, network partitions, and node failures. We approach this task by reexamining the notion of concurrency control and the intended purpose of a DLM in a shared-disk cluster.

2.3 Our model of distributed computation

We represent computation in a shared-disk cluster using the following abstract model: We consider a fully asynchronous distributed environment, in which processes run at different speeds and communicate via message passing over an asynchronous network. We assume a reliable FIFO channel for pairwise communication, but a message may take arbitrarily long to reach its destination. A set of network-attached disk devices provides persistent storage for application state and clients access this state by sending *Read* and *Write* requests to respective disk targets. Each disk device stores some number of *logical resources*, which represent application-level entities and are uniquely identified by a *resourceID*. Each resource resides on precisely one storage device, denoted its *owner*.

Application processes assume the existence of a DLM service and rely on it to coordinate concurrent access to the set of shared resources. The DLM provides shared-exclusive locking via the following two operations:

$$UpgradeLock(resourceID, fromMode, toMode)$$

$$DowngradeLock(resourceID, fromMode, toMode).$$

The three allowable lock modes are *NoLock*, *Shared*, and *Excl* and we assume that clients' interactions with the DLM are well-formed in the following sense:

When holding a lock in *NoLock* mode, a client may request an upgrade to *Shared* or *Excl* modes.

When holding a lock in *Shared* mode, a client may request an upgrade to *Excl* or a downgrade to *NoLock*.

When holding a lock in *Excl* mode, a client may request a downgrade to *Shared* or *NoLock* modes.

A *Shared* lock on resource *R* conflicts with every *Excl* lock on *R* and an *Excl* lock conflicts with every *Shared* and *Excl* lock on the same resource. In addition, if a lock is granted to some process c_1 and a conflicting lock is requested by another process c_2, the DLM service may issue a $RevokeLock(resourceID, toMode)$ notification to c_1, which can be considered a hint that c_1's current lock ownership on the respective resource is blocking another client's progress.

Any process (including components of the DLM service) may fail by crashing, but we do not consider target device failures in this report, since those can be handled using traditional techniques such as hardware-level redundancy [20] or application-level replication [21].

3 Design

3.1 Approach overview

At a high level, our approach reexamines the correctness criteria that a cluster DLM service must provide to applications. Traditionally, DLMs tend to treat shared application resources as purely abstract entities and achieve coordination by enforcing the *group mutual exclusion* property: no two client processes may simultaneously hold conflicting locks on the same shared resource. We note, however, that the mutual exclusion property as stated above is provably unattainable in an asynchronous system that is subject to even a single crash failure - a consequence of the impossibility of consensus [22] in such an environment. Furthermore, a hypothetical lock service that does offer such guarantees would not by itself suffice to guarantee data safety in such a setting, as Scenario 2 in the previous section suggests.

[1] "Shoot The Other Machine In The Head"

Rather than restricting access to a critical section of application code, our approach views the access coordination problem in terms of I/O request ordering guarantees that the storage system must provide to application processes. We refer to this alternative notion of correctness using the term *session serializability*, which we now specify formally. We begin by defining the notion of a *session* to a particular shared resource on disk:

Definition 1. *If a client process* `c` *issues a request* `UpgradeLock(R,..., Shared)` *to the lock service for some shared resource* `R` *and receives a positive acknowledgment, we say that* `c` *establishes a* ***Shared*** *session to* `R`*. An existing* ***Shared*** *session is terminated when* `c` *issues a request* `DowngradeLock(R,..., NoLock)`*. Analogously, by calling* `UpgradeLock(R,..., Excl)` *a process establishes an* ***Exclusive*** *session to* `R` *that can subsequently be terminated by downgrading to* ***Shared*** *or* ***NoLock****.*

For a given point `t` *in the execution history, we define* `S(t,c,R)` *to be the set of* `c`*'s active sessions to* `R` *at time* `t`*, determined solely by the sequence of prior* `UpgradeLock` *and* `DowngradeLock` *requests submitted to the DLM service.* `S(t,c,R)` *may contain a* ***Shared*** *or an* ***Exclusive*** *session to* `R`*, or both, or none.*

We say that a ***Shared*** *session to* `R` *conflicts with every* ***Exclusive*** *session to the same resource and an* ***Exclusive*** *session conflicts with every other session on the same resource.*

Definition 2. *If an I/O request* `r` *on a shared resource* `R` *is issued by a client process* `c` *at time* `t`*, we say that* `r` *is submitted* ***as part of*** *some session* `s` *to* `R` *if* $s \in S(t,c,R)$*. For a given session* `s` *to some resource* `R`*, we additionally define* **`Requests(s)`** *to be the set of all I/O requests on* `R` *submitted by the client as part of* `s`*.*

Definition 3. *A given execution history* `H` *is* ***session serializable*** *with respect to a shared resource* `R` *if the sequence of I/O request messages* $M = \langle r_1, r_2, \ldots \rangle$ *observed and processed by* `R`*'s owner satisfies:*

$$\forall r_i, r_j \in M \text{ such that } \{r_i, r_j\} \subset Requests(s) \text{ for some } s:$$

$$\nexists r_k \in M \text{ such that } i < k < j \text{ and } r_k \in Requests(s^*)$$

for some s^* *that belongs to another client and conflicts with* s*.*

Informally, the above invariant specifies that R's owner disk must observe prefixes of all sessions to R in strictly serial order, ensuring that no two requests in a session are interleaved by a conflicting request from another client. To illustrate this definition, consider a pair of concurrent request sequences from two clients shown in Figure 1. In this example, c_1 first performs two *Read* operations on X under the protection of a *Shared* lock, then upgrades to an *Excl* lock and performs two $Write$s and lastly, downgrades to *Shared* and performs two more *Read*s. Client c_2 acquires a *Shared* lock on X and submits a *Read* request, followed by an upgrade to *Excl* and two $Write$ requests. In this scenario, the following two sequences of request observations at X would be consistent with session serializability:

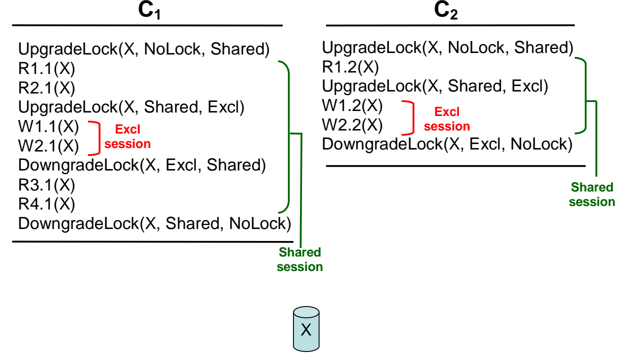

Figure 1: Concurrent request streams to a shared resource X from two application clients, c_1 and c_2. $R_{i.j}$ denotes the i-th *Read* operation from client j and, $W_{i.j}$ represents a *Write* operation, accordingly.

$$S_1 = \langle R_{1.1}, R_{2.1}, W_{1.1}, W_{2.1}, R_{3.1}, R_{4.1}, R_{1.2}, W_{1.2}, W_{2.2} \rangle$$
$$S_2 = \langle R_{1.1}, R_{2.1}, W_{1.1}, R_{1.2}, W_{1.2}, W_{2.2} \rangle$$

An execution history that causes X to observe $\langle R_{1.1}, R_{1.2}, R_{2.1}, W_{1.1}, W_{1.2} \rangle$ is not session serializable because it interleaves $W_{1.1}$, an exclusive session request from c_1, with two shared session requests from c_2: $R_{1.2}$ and $W_{1.2}$.

Note that session serializability is more permissive than strict mutual exclusion and in particular, permits execution histories in which two clients simultaneously hold conflicting locks on the same shared resource. At the same time, one could argue that these semantics meaningfully capture the essence of shared-disk locking, by which we mean that the request ordering guarantees in our model are precisely those that applications developers have come to expect from a traditional DLM service in such environments. Returning to the example of Figure 1, a conventional locking scheme that grants clients' requests in the order $\langle c_1(Shared), c_1(Excl), c_2(Shared), c_2(Excl) \rangle$ would cause X to observe S_1, while S_2 would correspond to the following scenario:

1. *Shared* lock on X is granted to c_1.
2. c_1 executes $R_{1.1}$ and $R_{2.1}$.
3. *Excl* lock on X is granted to c_1.
4. c_1 executes $W_{1.1}$

5. c_1 crashes and its *Shared* and *Excl* locks are reclaimed.

6. *Shared* lock on X is granted to c_2.

7. c_2 executes $R_{1.2}$.

8. *Excl* lock on X is granted to c_2.

9. c_2 executes $W_{1.2}$ and $W_{2.2}$.

Minuet focuses on ensuring safe ordering of I/O requests at storage devices consistent with session serializability and explicitly avoids enforcing global agreement on the state of locks and group membership views. As a result, our design does not necessitate the use of a complex and expensive agreement protocol and does not impose the associated limitations on availability.

The basic idea behind our approach is to augment the shared disk device with a small amount of application-independent logic, which we call a *guard*, that enforces the session serializability invariant on the stream of incoming I/O requests. Minuet associates a *session identifier* (SID) with every lock instance granted to a client and we modify the disk I/O protocol stack at the client side to annotate all outgoing I/O requests with client's current *SID* for the respective resource. Below, we refer to this annotation as a *request capsule*.

The guard logic at target disk devices evaluates incoming I/O requests based on the attached *SID* and, for each request, determines whether its acceptance would violate session serializability. All such requests are dropped from the input stream and the originating client is notified via a special error code *EBADSESSION*. From an application developer's point of view, session rejection appears as a failed I/O request and an exception event notification from the lock service indicating that a particular lock is no longer valid. This may require the application to take a corrective action, such as discarding the respective object from local cache buffers, rolling back any associated changes and, possibly, retrying the previous operation after reacquiring the lock under another *SID*.

The guard logic situated at I/O target devices addresses the safety problems due to delayed messages and inconsistent failure observations that plague asynchronous distributed environments and enforcing safety at the target device allows us to simplify the core functionality of the DLM module. In Minuet, the primary purpose of the lock service is ensuring a consistent and efficient assignment of session identifiers to clients in a manner that minimizes the aggregate rate of session rejection in the cluster.

Decoupling correctness from performance in this manner enables substantial flexibility in the choice of mechanism used to control the assignment of session identifiers. At one extreme is a purely optimistic technique, whereby

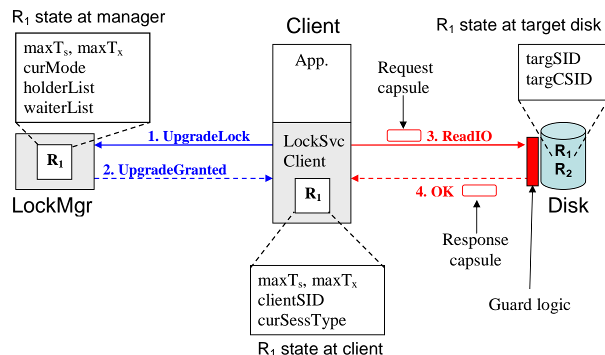

Figure 2: Protocol messages and per-resource state at application clients, lock managers, and shared disks.

every client selects its *SID*s via an independent local decision, without attempting to coordinate with the remainder of the cluster and this might be an entirely reasonable strategy for applications and workloads characterized by a consistently low rate of data contention. A traditional DLM service that serializes all session requests at a central lock server can be viewed as a design point at the other extreme. Minuet tries to position itself in the continuum between these extremes in order to allow application developers to trade off lock service availability, synchronization overhead, and I/O performance under heterogeneous data access patterns.

Next, we describe the protocol machinery for enforcing session serializability on a single shared resource and then demonstrate how more complex and useful application semantics, such as distributed transactions, can be supported using session serializability as a foundational building block. Lastly, we address the issue of fault tolerance and present a quorum-based algorithm for loosely-consistent replication of lock management state.

3.2 Enforcing session serializability

We use a simple timestamp-based mechanism to enforce session serializability semantics on an individual shared resource. A client's session to a given resource R is identified by a value pair $\langle T_s, T_x \rangle$ specifying a *shared* and an *exclusive* timestamp, respectively. To acquire a lock on R, a client first *proposes* a session timestamp to the lock manager. These proposal are globally unique - no two clients propose an identical pair of values and no client proposes the same value pair twice. Our current design accomplishes this via the following timestamp format: $\langle T.cliID.incNum \rangle$, where *cliID* uniquely identifies a client process and *incNum* is client's incarnation number - a monotonic counter used to ensure uniqueness across crashes.

The basic locking protocol proceeds as follows: each client c maintains an estimate of the largest session timestamp previously granted to any client, which we

denote $MaxT_s(c,R)$ and $MaxT_x(c,R)$. To acquire a *Shared* lock on R, client proposes a new session timestamp $\langle ProposedT_s, ProposedT_x \rangle$, where $ProposedT_x = MaxT_x(c,R)$ and $ProposedT_s$ is the smallest unique timestamp greater than $MaxT_s(c,R)$.

Client then sends an $UpgradeLock$ request to the lock manager, specifying the desired mode (*Shared*) and the proposed timestamp pair. The lock manager accepts and enqueues this request if no request with a larger $ProposedT_x$ value has previously been accepted. Otherwise, the manager denies the request and responds immediately with $UpgradeDenied$, which specifies the largest timestamp values previously observed by the manager. In the latter case, the client updates its local estimates $MaxT_s(c,R)$ and $MaxT_x(c,R)$ and submits a new proposal. After accepting and enqueuing c's request, the lock manager eventually grants it and responds with a $LockGranted$ message. Upon receipt of this message, the client sets $cliSID(R) = \langle ProposedT_s, ProposedT_x \rangle$ and $sType(R) = Shared$.

Acquisition of an *Exclusive* lock (which includes upgrading from *Shared* to *Excl*) proceeds analogously except that clients increment the T_x value in the proposed session ID and the lock manager checks both $ProposedT_s$ and $ProposedT_x$ when determining whether to enqueue or deny the request. After receiving a $LockGranted$ response, the requesting client sets the session type ($sType$) to $Excl$.

When issuing a $Read$ or a $Write$ request to a disk target D, the application specifies the ID of the shared resource affected by the request. Before transmitting the request to D, we augment it with a *request capsule* that encodes client's session identifier for the specified resource in a tuple of the following form: $\langle resourceID, sType, cliSID, curCSID, nextCSID \rangle$. (The last two parameters specify *commit session identifiers* - an additional piece of state used for supporting transactional updates and we describe its purpose in Section 3.3.1).

For every shared resource R owned by a target disk device D, the device maintains the *target session identifier* ($targSID(R)$). Upon receipt of an I/O request from a client, D examines the request capsule and looks up the $targSID$ entry for the specified resource ID, evaluates the request capsule against the current entry, processes the I/O request, and sends a response as shown in Algorithm 1. Figure 2 illustrates the basic message exchange between a client, a lock manager, and a storage device for a simple read operation under a *Shared* lock, as well as locking-related state maintained at each component.

If client receives $EBADSESSION$ in response to its I/O request, Minuet examines the response capsule and notifies the application process that its lock on the respective resource is no longer valid. Specifically, a lock held previously in $Excl$ mode is downgraded to $Shared$ if $targSID.T_s > cliSID.T_s$ and a $Shared$ lock is further downgraded to $NoLock$ if $targSID.T_x > cliSID.T_x$. The client also updates $MaxT_s(c,R)$ and $MaxT_s(c,R)$ to reflect the most recent timestamp values specified in the response capsule and informs the lock manager that a *forced downgrade* has taken place.

Algorithm 1 Guard logic at a SAN target device ($reqCaps$ denotes the request capsule sent by a client along with the respective I/O request).

$resourceID \leftarrow reqCaps.resourceID$
$\langle targSID, targCSID \rangle \leftarrow LookupState(resourceID)$
if $(targSID \neq NIL)$ **then**
  $decision \leftarrow \underline{REJECT}$
  if $(targCSID = reqCaps.curCSID)$ **then**
    if $(reqCaps.sType = Shared)$ **then**
      if $(reqCaps.cliSID.T_x \geq targSID.T_x)$ **then**
        $decision \leftarrow \underline{ACCEPT}$
      end if
    else {Exclusive session}
      if $((reqCaps.cliSID.T_s \geq targSID.T_s) \wedge (reqCaps.cliSID.T_x \geq targSID.T_x))$ **then**
        $decision \leftarrow \underline{ACCEPT}$
      end if
    end if
  end if
else {Entry not found}
  $decision \leftarrow \underline{ACCEPT}$
end if
if $(decision = ACCEPT)$ **then**
  $targSID.T_s \leftarrow MAX(targSID.T_s, reqCaps.cliSID.T_s)$
  $targSID.T_x \leftarrow MAX(targSID.T_x, reqCaps.cliSID.T_x)$
  $targCSID \leftarrow capsule.nextCSID$
  $UpdateState(resourceID, \langle targSID, targCSID \rangle)$
  $rc \leftarrow ProcessIORequest()$
  $respCaps \leftarrow NIL$
else
  $rc \leftarrow EBADSESSION$
  $respCaps \leftarrow \langle resourceID, targSID, targCSID \rangle$
end if
Send $\langle rc, respCaps \rangle$ to client

Claim 1. *For every shared resource R, the locking protocol and capsule evaluation mechanism described above guarantees session serializability. (The proof can be found in Appendix A).*

3.3 Supporting transactional semantics

3.3.1 Overview

Transactions, defined by Gray [23] as transformations of state having the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation), are widely regarded as a useful pro-

gramming primitive. In distributed shared-disk applications, transactional semantics are typically supported by a two-phase locking protocol for isolation and a write-ahead logging facility (in some cases referred to as *journaling*) to achieve update atomicity and durability. Commonly, in order to commit a transaction, an application process appends to the log a sequence of Redo records that concisely describe the set of modifications performed by the transaction, after which a special *Commit* record is force-appended to the log. Prior to releasing a lock on a dirty resource *R*, its holder must sync all committed updates to the snapshot of *R*, so as to ensure that the next reader observes the effects of every committed transaction. If a client process fails while holding locks on a non-empty subset of resources, the system enters a recovery phase, during which a (potentially distributed) recovery process examines the faulty client's portion of the log, determines the set of transactions with operations that need to be replayed or rolled back, and performs the corresponding operations in order to restore the set of affected resource snapshots to a consistent state.

At a high level, our approach tries to borrow from state-of-the art mechanisms for transaction support to the largest extent possible, while introducing several extensions to address their liveness and safety limitations in an asynchronous setting. Since the primary focus of our work is feasibility rather than performance optimization, our current design provides only a subset of features typically found in a state-of-the-art transaction service such as D-ARIES [10]. Below, we present a design that implements redo-only logging to support the "no force no steal" buffer policy[2] and currently, our design permits only one active transaction at a time - after starting a transaction, a client must commit or abort before starting the next transaction. Finally, we assume unbounded log space for each client. These restrictions allow us to focus the discussion on the novel aspects of our approach and we believe that additional optimizations, such as support for Undo logging, can be retrofitted onto the scheme presented here in a relatively straightforward manner. Below is a list of principles and requirements that guided our design:

(1) Eliminate reliance on strongly-consistent locking. Rather than requiring all clients to coordinate concurrent activity through a strongly-consistent DLM service, the session serializability logic we added to the storage devices provides a limited form of transaction coordination and allows us to relax the degree of consistency required from the DLM. Before committing (or completing a read-only transaction), we require clients to verify that all locks acquired at the start of the transaction are still valid - we call this step the *verification phase*. During this phase, client issues I/O requests to all storage devices that hold resources touched by the transaction and proceeds to committing only if it collects a positive response for every resource, indicating that its sessions are still valid. This mechanism allows us to identify and resolve cases of conflicting access from multiple clients due to inconsistent locking state and can be seen as a variant of optimistic concurrency control - a well-known technique in database design [24].

(2) Avoid enforcing a globally-consistent view of process group membership. Rather than relying on a group membership service to detect client failures and initiating recovery proactively in response to perceived failures, our design explores a *lazy* or *on-demand* approach to transaction recovery that tries to postpone the recovery action until the affected data is needed, which permits the system to function without global agreement on group membership. As we demonstrate below, keeping a small amount of additional per-resource state at a storage device allows us to detect cases when an incoming I/O request would touch a potentially inconsistent piece of data (e.g., missing some updates from a committed transaction). All such requests are rejected by the storage device with the *EBADSESSION* response code. Upon receiving this response, an application process may choose to reacquire its lock on the respective resource which, under normal conditions, would cause its current holder (c_h) to observe a *RevokeLock* request, sync updates to the snapshot, and eventually downgrade the lock. Alternatively, upon suspicion that c_h has crashed, the process may initiate a recovery action and attempt to bring the snapshot back to a consistent state by reapplying missing updates from the log. Crucially, the choice between *coordinating* and *initiating failure recovery* is a local decision and the integrity of application state on disk is not dependent upon correctness of the failure observation.

(3) Avoid introducing assumptions of synchrony typically required for log-based recovery in a shared-disk setting. The session serializability disk extension proposed in Section 3.2 enables a target disk to detect and reject I/O requests that conflict with log recovery and would otherwise result in violation of safety. For instance, a delayed $Write$ command from a faulty client c that carries an update to some resource R would get rejected by R's owner device if another client has already opened c's log and started reapplying missing updates to the image of R under a session with a larger timestamp.

3.3.2 Basic transaction protocol

Our current design uses a set of per-client logs on shared disks to record transaction redo information. These logs appear to clients as regular lockable resources that can be read and written to, while the underlying storage device

[2] *No force* means that committing a transaction does not require flushing all updates to the snapshot - it suffices to write a commit record to the log. *No steal* implies that dirty data buffers containing updates from *uncommitted* transactions cannot be flushed to stable storage.

is assumed to provide session serializability guarantees in cases of concurrent access from multiple clients. The physical location (i.e., disk identifier and starting offset) of a client's log is easily computable from the client process identifier (*cliID*).

To support distributed transactions, we extend the basic session serializability machinery introduced in Section 3.2 with an additional piece of state, which we call a *commit session identifier (CSID)*, of the form $\langle cliID, xactID \rangle$. For each shared resource R, its owner device maintains the $targCSID(R)$ value alongside $targSID(R)$. Conceptually, the value of $targCSID(R)$ at a particular point in the execution history identifies the most recent transaction that may have updated R and committed without syncing the corresponding changes to disk. If $targCSID(R) = NIL$, the image on disk is guaranteed to be consistent with the set of committed transactions and can thus be safely accessed. Conversely, $targCSID \neq NIL$ indicates that R's current state on disk might be missing some updates from a committed transaction and therefore cannot be assumed valid. In this case, the *cliID* portion of *targCSID* can be used to locate the log of the client responsible for these modifications, identify the commit status of relevant transactions, and restore consistency of R by reapplying the missing updates from the log. This is the basis of the lazy recovery mechanism, which we describe more fully in Section 3.3.4.

Algorithm 2 illustrates transaction execution in high-level pseudocode. Unless specified otherwise, every I/O request to a remote disk affecting a resource R carries the following state in its request capsule:

$$\begin{aligned} &\langle resourceID, sType, SID, curCSID, nextCSID \rangle \\ =&\langle R, R.sType, R.cliSID, \\ &\langle cliID, R.xactID \rangle, \langle cliID, R.xactID \rangle \rangle \end{aligned}$$

During initialization, a client process c acquires an exclusive lock on its own log resource (denoted $c.Log$) and reads its content from disk, thereby establishing an exclusive session to the log. To begin a new transaction T, c increments the active transaction identifier *activeXactID*, a monotonically increasing counter, and appends a *BeginXact* record to its log. Next, in the Read phase of the transaction, c acquires *Shared* locks on all resources in the *ReadSet* of T and reads the corresponding data from disk into local buffers[3]. In the *update phase* that follows, client performs the desired set of update operations on resources in its $WriteSet$ (at this stage without forcing locally-buffered data to disk) and appends the corresponding set of $Update$ records to its log. Each such record describes a single write I/O operation that modifies a contiguous region of data on disk and is of the form $\langle R, D(R), offset, len, data \rangle$, where R is the resource being updated and $D(R)$ is the disk on which R resides.

Next, transaction proceeds to the *verification phase*, during which the client confirms the validity of its sessions for all resources touched by the transaction (and hence, the accuracy of cached data). Algorithm 3 specifies the *VerifySession* function in high-level pseudocode. For every resource in the *ReadSet*, the client contacts its target device and verifies that its current session ID is still valid and that no conflicting transaction has attempted to commit[4]. For elements of the $WriteSet$, in addition to verifying that the exclusive session identifier is valid, the client also prepares them for committing the active transaction by specifying $\langle cliID, activeXactXID \rangle$ in the *nextCSID* field of the request capsule. As shown in Algorithm 1, if the request successfully passes verification at the target device, the *targCSID* value is set to the *nextCSID* field of the request capsule. As a result, a subsequent attempt by another process to access some $WriteSet$ element R would fail and observe a non-*NIL* *targCSID* value in the response capsule and that process would either wait for the local process to flush the corresponding updates to the image of R or initiate a recovery action and update R from local client's log.

Note that the extra round of communication added by the verification phase is the penalty our approach pays for the absence of strict coordination provided by a strongly-consistent DLM. Following successful verification, the transaction enters the *commit phase*, in which the client appends a *CommitXact* record to its log and forces the log tail to disk.

The basic protocol outlined above provides transaction isolation, identifying cases of conflicting access during the verification phase. However, recall that under our weakly-consistent model of locking and session serializability semantics, any disk I/O operation (including access to a client's log) may fail with *EBADSESSION* due to a conflicting access from another client. This leads to several additional exception cases at various stages of transaction execution, which are shown in Algorithm 2 (numbers 1 through 5) and are briefly discussed below.

Case (1): Client loses a session to its log while trying to read it during initialization and receives *EBADSESSION* and a forced downgrade to *NoLock*. In this case, the client simply reacquires the lock on its log resource under another SID and retries, repeating the process if necessary.

Case (2): During the *Read* phase, an attempt to read some resource R from disk is rejected with *EBADSESSION* and a forced downgrade to *NoLock*. Client recovers by aborting and restarting the current transaction. Note that if the response capsule supplied by

[3]Of course, locks and memory buffers holding cached copies of shared resources can be retained across transactions for efficiency.

[4]Our current design implements Verify I/O requests as zero-length Reads and Writes, whose sole purpose is to transport a capsule to the target device for verification.

the target disk indicates $targCSID \neq NIL$, the image of R on disk may be missing some committed updates and if the client process specified by $targCSID.cliID$ is suspected to be faulty, the local client may choose to initiate log recovery and restore R to a consistent state, using the protocol presented in Section 3.3.4.

Cases (3) and (4): A verify request is rejected with $EBADSESSION$ and a forced downgrade to $SharedLock$ or $NoLock$. We treat this scenario analogously to case (2): client aborts the transaction and performs recovery on the set of resources that failed session verification and whose response capsules specify a non-NIL $targCSID$ value.

Case (5): Transaction passes the verification phase and successfully updates $targCSID$ on all resources in the $WriteSet$, but forcing a $XactCommit$ record fails due to loss of session to the log. In this scenario, the active transaction cannot be committed since another client may have chosen to initiate log recovery for the local client, read its log, and abort this transaction. The local client aborts the active transaction and may restart it after reestablishing an exclusive session to its log.

3.3.3 Syncing updates to disk

Since our current design provides redo-only logging, a dirty memory buffer holding a modified copy of some resource R may be flushed to disk only if every prior transaction that modified R has been successfully committed. Typically, a client process would write back its buffered copy of R to disk upon receiving a lock revocation request on R from the DLM service.

To sync a modified copy of resource R, a client simply writes out the local buffer to the target device, issuing a sequence of one or more disk $Write$ requests and specifying the following parameters in the request capsule:

$$\begin{aligned} &\langle resourceID, sType, SID, curCSID, nextCSID\rangle \\ =&\langle R, Excl, R.cliSID, \langle cliID, R.xactID\rangle, \\ &\langle cliID, R.xactID\rangle\rangle \end{aligned} \tag{1}$$

The last request in the sequence specifies $nextCSID = NIL$ and, upon receiving and processing this request, the target device resets its $targCSID$ to NIL, which effectively marks the disk image of R as "clean" for the next reader. After completing this step, an $UpdateSynced$ record of the form $\langle R, R.xactID\rangle$ is appended to the log, indicating that the image of R on disk has been updated to reflect the effects of all transactions up to $R.xactID$. A committed transaction T can be purged from the log, and the corresponding space reclaimed, if for every element R in T's $WriteSet$, an $UpdateSynced$ record $\langle R, R.xactID\rangle$ satisfying $R.xactID \geq T.xactID$ has been added to the log.

If the sync operation fails to complete due to loss of session with R, the client simply invalidates the cached buffer and no additional actions need to be taken.

Algorithm 2 Basic transaction protocol

```
Start the transaction service:
(One-time initialization)
  UpgradeLock(c.Log, NoLock, Excl)
  ReadFromDisk(c.Log)                                        (1)
  curXactID ← largest XID found in the log
Begin Transaction:
  curXactID ← curXactID + 1
  LogAppendRec(⟨ BeginXact, curXactID ⟩)
Read Phase:
  for all resources R in ReadSet do
    UpgradeLock(R, NoLock, Shared)
    ReadFromDisk(R) (into local buffer)                      (2)
  end for
Update Phase:
  for all resources R in WriteSet do
    UpgradeLock(R, NoLock, Excl)
    for all update operations U on R do
      Apply U to local copy of R
      LogAppendRec(⟨Update, U⟩)
      ReadSet ← (ReadSet − {R})
    end for
  end for
Verification Phase:
  RejectSet ← ∅
  for all resources R in ReadSet do
    rc ←VerifySession(R, Read)                               (3)
    if (rc = EBADSESSION) then
      RejectSet ← RejectSet ∪ {R}
    end if
  end for
  for all resources R in WriteSet do
    rc ← VerifySession(R, Write)                             (4)
    if (rc = EBADSESSION) then
      RejectSet ← RejectSet ∪ {R}
    end if
  end for
Completion Phase:
  if (RejectSet = ∅) then
    if (WriteSet ≠ ∅) then
      LogAppendRec(⟨CommitXact⟩)
      ForceTailToDisk(c.Log)                                 (5)
      for all resources R in WriteSet do
        R.xactID ← activeXactID
      end for
      return COMMITTED
    else {Read-only transaction}
      return COMPLETED
    end if
  else
    return ABORTED
  end if
```

Algorithm 3 Function $VerifySession(R, mode)$

$curCSID \leftarrow \langle cliID, R.xactID \rangle$
if $(mode = Read)$ **then**
  $nextCSID \leftarrow \langle cliID, R.xactID \rangle$
  $reqCaps \leftarrow \langle R, R.sType, R.cliSID, curCSID, nextCSID \rangle$
  $rc \leftarrow ReadFromDisk(R, reqCaps)$ {Zero-length read}
else
  $nextCSID \leftarrow \langle cliID, activeXactID \rangle$
  $capsule \leftarrow \langle R, R.sType, R.cliSID, curCSID, nextCSID \rangle$
  $rc \leftarrow WriteToDisk(R, reqCaps)$ {Zero-length write}
end if
return rc

3.3.4 Lazy transaction recovery

If an I/O request from a client c to a shared resource R fails with *EBADSESSION* and if a non-*NIL* *targCSID* value is specified in the response capsule then the image of R on disk may be missing some committed updates. In this case, *targCSID.cliID* identifies the client process responsible for these updates and *targCSID.xactID* specifies the most recent transaction in the respective client's log that may have updated R. If c suspects that client to be faulty, it may initiate a recovery action that brings the disk image of R up to date and proceeds as follows:

The recovery process acquires an exclusive lock on *targCSID.cliID.Log* and reads its content from disk. It searches the log for the most recent *UpdateSynced* record for resource R and sets *MaxSyncedXactID* to the *xactID* field of that record. Next, the client identifies the set of committed transactions with $XactID \geq MaxSyncedXactID$ that include one or more *Update* records for resource R and aggregates these updates into a *redo operation list*. After acquiring an exclusive lock on R and obtaining the corresponding session identifier (*R.cliSID*), the client reapplies the sequence of operations in the redo list by issuing corresponding write requests to the target disk. The following set of parameters is specified in the request capsule:

$$\langle resourceID, sType, SID, curCSID, nextCSID \rangle = \langle R, Excl, R.cliSID, targCSID, targCSID \rangle$$

The last request in the sequence specifies $nextCSID = NIL$, which causes the target device to reset *targCSID* to *NIL*, thereby indicating to the next reader that the disk image of R has been brought up to date. As the last step, the recovery process force-appends $UpdateSynced\langle R, targCSID.xactID \rangle$ to faulty client's log[5].

Due to loosely-consistent locking, the recovery process may experience loss of its exclusive sessions to the log or the actual resource being repaired. This may happen, for instance, if a remote client initiates a concurrent recovery action on R and succeeds in reapplying some or all of the missing updates, thus breaking local client's sessions. In both cases, it is safe to simply abort the recovery operation and reattempt the application-level action.

3.4 Lock manager replication

The manager component of a cluster lock service can be replicated for fault tolerance and typically, strongly-consistent replication is needed to provide the desired coordination semantics. In our model, the DLM service is not required to guarantee full mutual exclusion; instead, the goal is to provide some limited form of coordination that enables efficient access to data and minimizes the rate of I/O rejection for a given application workload. This enables a simpler and generally more available replication design that allows clients to retain progress in the face of extensive node and connectivity failures. For instance, our design does not require connectivity to a majority of manager processes - a lock can be acquired as long as at least one of the manager instances is reachable [6].

To support manager replication, we extend the basic locking protocol presented in Section 3.2 as follows: When acquiring or upgrading a lock, client selects a subset of managers, which we call its *request quorum*, and sends an *UpgradeLock* request with the corresponding timestamp proposal to all members of this set. The lock is considered granted (and the application is notified) once an *UpgradeGranted* response is collected from all quorum members. If any of the members respond with *UpgradeDenied* due to an outdated timestamp in the request, the client downgrades the lock on all members that have previously responded with *UpgradeGranted*, then updates its $MaxT_s$ and $MaxT_x$ values, and resubmits the upgrade request with a new timestamp proposal. For efficiency, we allow *UpgradeLock* requests to specify an *implicit downgrade* for an earlier timestamp, which permits us to combine these two requests into a single message.

Incoming revocation requests from quorum members that have responded with *UpgradeGranted* are buffered until the client hears back from the entire quorum. Eventually, if the upgrade is granted by all members, the application process is notified and any pending revocation requests are also delivered at that time.

To illustrate, consider a basic scenario with two clients (c_1 and c_2) and two manager processes (m_1 and m_2) and

[5]Note that as an optimization, in addition to recovering the state of resource R that initially triggered the recovery action, we can attempt to repair the missing updates for all resources touched by the faulty client from the log, but this is not strictly necessary. In principle, any resource can be recovered in an analogous lazy manner after a rejected access attempt.

[6]In an extreme case, that instance can be the local DLM client itself, which would simply grant its own proposals without coordinating with other clients.

suppose these clients make concurrent attempts to acquire an exclusive lock on some resource R. Suppose the timestamp proposals are $\langle 0,1 \rangle$ and $\langle 0,2 \rangle$ for c_1 and c_2, respectively, and suppose their upgrade requests reach the two managers in opposite orders. m_1 first observes the request from c_1 and grants the lock immediately and later, upon receiving c_2's request with a higher timestamp, accepts it and places c_2 on the queue of waiters. m_2 first observes the $UpgradeLock$ message from c_2 and grants the lock immediately and when c_1's request arrives, this manager denies it because c_2's proposal with a higher timestamp has been observed. Upon receiving $UpgradeDenied$ from m_2, c_1 selects a new timestamp proposal, say $\langle 0,3 \rangle$, and sends another upgrade request to both managers, additionally specifying an implicit downgrade on $\langle 0,1 \rangle$. When m_1 receives this message, it honors the downgrade request, grants the lock to the next waiter - c_2, and places c_1 on the queue of waiters because its timestamp proposal now supercedes c_2's. Likewise, m_2 accepts c_1's request and enqueues it. At this point, c_2 has been granted locks by both managers and can proceed to issuing I/O requests on R under session ID $\langle 0,2 \rangle$. When c_2 completes its operation on R, it sends a downgrade request to both managers. They both grant the lock to c_1 - the next waiter, which then proceeds to accessing R under SID $\langle 0,3 \rangle$ and thus, proper serialization is achieved.

The basic scheme suggested above is by no means the only feasible replication mechanism for a loosely-consistent lock service and a number of obvious optimizations can be considered. For example, lock managers can coordinate among themselves and disseminate changes to the list of holders and waiters in a lazy manner and in order to reduce the frequency of $UpgradeDenied$ responses, clients can gossip about the maximum known timestamp for each resource. We hope to explore and evaluate some of these optimizations in future work.

4 Implementation

We have implemented a proof-of-concept prototype of Minuet based on the design presented in the preceding section along with several sample parallel applications. The prototype has been implemented on the Linux platform in C and the implementation consists of a client DLM library, a lock manager process (7630 LoC), a storage process that emulates a SAN target device (630 LoC), and sample applications (920 LoC).

4.1 Client-side lock service library

The client-side component of Minuet is implemented as a statically-linked library and offers C language bindings for client applications. It is based on an asynchronous event notification mechanism and provides a basic locking service, an I/O interface to remote disks, and a transaction service. The core elements of the application interface are illustrated in Algorithm 4.

Algorithm 4 Minuet client library API

Basic lock service:
UpgradeLock(resID, upgradeMode, coordFactor)
DowngradeLock(resID, downgradeMode)
Shared disk I/O:
DiskRead(diskID, resID, offset, length, dataBuf)
DiskWrite(diskID, resID, offset, length, dataBuf)
Transaction service:
BeginXact()
AddUpdate(resID, diskID, offset, length, data)
AbortCurXact()
CommitCurXact(readset, writeset)
MarkResourceSynced(resID, xactID)

When issuing an $UpgradeLock$ request for some resource R, an application optionally specifies the *coordination factor* (C), which determines the size of the lock manager quorum Q, i.e., the number of manager processes that must agree to grant the requested lock on R. Currently, the quorum size is computed as follows: $Q = \lceil C\frac{M}{2} + 1 \rceil$, where M is the total number of lock manager replicas.

This parameter allows application developers to tune the degree of locking consistency provided by the DLM, enabling a choice between optimism and strict coordination and a tradeoff between availability and synchronization overhead. A small quorum size works well for low-contention resources; it helps keep the lock message overhead low and permits clients to make progress in a partitioned network, but exposes application clients to I/O rejection and forced downgrades in the event of conflicting access. Conversely, a large quorum reduces the probability of rejection, but requires connectivity to a larger number of manager replicas. If every application process specifies a coordination factor of 1 (i.e., a majority quorum) for every lock acquisition request, our system would effectively behave as a traditional strongly-consistent DLM. When the upgrade request is granted by a quorum, an $UpgradeGranted$ notification is posted to the application event queue.

The client-side DLM module maintains a heartbeat session with each manager process (currently implemented via TCP keepalive) and a local estimate of its liveness. If a client succeeds in acquiring a lock on some resource R from a quorum of lock managers and the last remaining manager in that quorum crashes, a *SessionExposed* notification is posted to the application event queue. This event informs the application that all records of its lock (and the corresponding session) on R may have been lost and a subsequent attempt by another client to establish a session on R may be granted immediately and thereby cause loss of session at the local client. In order to prevent this, the

application can issue another *UpgradeLock* request and attempt to reacquire the lock under the same *SID* from another quorum. This mechanism helps reduce the amount of I/O rejection occurring as result of a manager failure and is particularly useful for protecting long-running sessions (e.g., fetching a large data structure from disk into local memory) that are costly to redo.

Functions *DiskRead* and *DiskWrite* provide an interface for submitting I/O requests to remote disks. We need to intercept these requests in the client library in order to augment them with resource session identifiers, as described in Section 3.2. For each I/O request, the resource being accessed is specified by the application as a function argument. The DLM client retrieves the corresponding session state and piggybacks it onto the request in the form of a capsule. When a response is received, an *ioCompletion* notification is posted to the application event queue. I/O requests rejected by the target disk return status code *EBADSESSION* and for all such requests, the DLM client additionally posts a *ForcedDowngrade* notification to inform the application that its lock on the respective resource has been downgraded to some weaker mode.

The transaction module implements the design of Section 3.3. Internally, it implements a log abstraction on top of raw disk and uses the basic lock service to ensure session serializable access to the log. Its application interface includes functions for initiating a transaction, logging an update, aborting, committing, and syncing updated resources to disk. The commit function requires the application to specify the *ReadSet* and *WriteSet* of the current transaction, which are examined during the verification phase. If verification fails due to loss of sessions, a corresponding set of *ForcedDowngrade* notifications is sent to the application. In addition, a commit operation may fail if the client loses its exclusive session to the log, in which case the application is notified via a *XactSvcFailure* event.

4.2 Lock manager process

Minuet's lock manager process is responsible for granting and revoking locks using the timestamp mechanism (Section 3.2) and several manager instances can be deployed in a cluster for fault tolerance. For each lockable resource R in the system, the manager maintains the current lock mode, a list of current holders, a queue of blocked upgrade requests, and the largest proposed timestamp values observed for R so far ($\langle MaxProposedT_s(R), MaxProposedT_x(R)\rangle$). When an *UpgradeLock* request on R arrives from some client c, the manager evaluates its proposed timestamps and either accepts it and adds c to the queue of waiters or responds with *UpgradeDenied*. When a new request incompatible with the current mode appears on the waiter queue, the manager attempts to revoke the lock from current holders by sending them a *RevokeLock* message, which in turn causes the application on these nodes to observe a *RevokeLock* event. A heartbeat mechanism is used to detect client failures and after a failure suspicion, the manager reclaims all of the locks previously held by suspected client and makes them available to the next waiter.

4.3 Storage process

Our current implementation emulates the functionality of a guard-augmented disk target via a user-level process that runs on a dedicated node, communicates with clients over a TCP socket, and writes data to a local disk partition or a file in a local filesystem. While this may not be an ideal representation of a SAN-attached disk, we were careful to preserve the semantics of a "dumb" storage device that supports only *Reads* and *Writes* on raw data blocks. Our metadata (*targSID* and *targCSID*) requires 16 bytes of memory per resource and the storage process currently maintains it in RAM using a hash table.

4.4 Sample applications

Distributed chunkmap Our first application implements a read-modify-write operation on a distributed data structure comprised of a set of fixed-length data chunks. It mimics atomic updates to a distributed chunkmap - a common scenario in clustered middleware such as filesystems and databases. The chunkmap may represent a bitmap of free space blocks (e.g., Scenario 1 in Section 2), an array of i-node structures, or an array of directory entry slots in a directory file. In each operation, the application process selects a random 4-KByte chunk, reads it from shared disk into a local buffer, modifies a randomly-selected region within the chunk, and writes it back to disk. Locking is used to ensure update atomicity: prior to reading the block from disk, the application process acquires an exclusive lock from Minuet on the respective block and releases it after writing the modified version to disk. In our evaluation, we measure the aggregate operation throughput from multiple clients under strong and loosely-consistent locking under varying levels of block contention.

Distributed transactional update To demonstrate the feasibility of serializable transactions (Section 3.3) in Minuet, we extend the basic chunkmap application described above with multi-block atomic update operations. In each iteration, the application process selects up to five distinct data blocks, acquires the respective locks, reads and updates their content, and attempts to commit these updates to disk in a single atomic action. This application exercises Minuet's write-ahead logging module and transaction API routines (*BeginXact*, *AddUpdate*, *CommitCurXact*, and *MarkResourceSynced*). If a transaction aborts due to loss of session to a datablock or the client's log, the application reacquires the lock on the

respective resource and retries the transaction (without backoff) until it commits successfully.

5 Evaluation

In the previous sections, we have shown how Minuet provides safety by adding guard logic to SAN target devices. In this section, we evaluate the performance and availability of the sample applications built atop Minuet and provide comparison with traditional strongly-consistent locking.

5.1 Experimental setup

We ran our experiments in an emulated SAN environment with 16 identical machines in our local cluster. Each node is a dual 3GHz Xeon machine equipped with 2 GByte RAM, two 7200 RPM IDE disks, and a Gigabit Ethernet NIC. The machines were interconnected via a Gigabit switch. We allocated five machines for storage servers and these nodes collectively provided 5 GB of logical disk space, equally striped across the servers. Three additional machines were assigned to serve as dedicated lock managers, each running a single instance of the Minuet manager process. Client instances were equally split across the remaining seven machines and they saturated neither RAM nor CPU. Note, however, that when evaluating the zero-degree locking consistency (i.e., the **weak-own** scenario described below), a lock manager process shared a machine with a matched client process.

In our experiments, we evaluated the performance of our applications under the following three scenarios:

strong(x): A strongly-consistent locking protocol that requires a client get permission from a majority (x) of lock manager processes.

weak-own: An extreme form of weakly-consistent locking, where each client acts as its own lock manager and does not attempt to coordinate with other clients.

weak-partition(x): Simulates a failure scenario, in which the network is partitioned into x distinct segments. Within a segment, Minuet provides strongly-consistent coordination through a single lock manager, but no coordination between segments takes place.

We measured the performance as the total number of application operations per second, varying the number of clients (i.e., offered load). We also considered two forms of workload, namely:

uniform: Each operation selects the block(s) to modify uniformly at random from the entire chunkmap.

skewed(x/y): y% of operations touch x% of the chunkmap.

We ran each experiment for 10 minutes and repeated it 8 times to compute the mean and standard deviation.

5.2 Distributed chunkmap

The distributed chunkmap application performs read-modify-write operations on an array of data blocks and relies on Minuet locking to ensure update atomicity. We configured the block size to 4 KByte and varied the number of client instances to evaluate the zero-consistency locking and the strong-consistency locking with different number of lock manager replicas.

Figure 3 shows the operation throughput, the denied lock requests at lock managers due to conflicting access, and the rejected I/O requests at SAN target devices with the *uniform* workload. Since there are a large number of blocks in the storage servers, these results represent a low-contention scenario. We observe that there is little performance difference between *strong(x)* and *weak-own* up to 16 clients, but in a high load case (32 clients) *strong(x)* shows considerably lower throughput than *weak-own*. This is because clients compete with each other to acquire locking state in lock manager processes and their requests are denied as shown in Figure 3 (center). On the contrary, *weak-own* does not incur locking overhead and scales throughput linearly, although a small fraction of clients' I/O requests is rejected at the disk (Figure 3 (right)). This result suggests that our approach is also beneficial in improving application throughput in scenarios where the overall load is high, but contention for a single resource is relatively rare.

We conclude that the current practice of enforcing full mutual exclusion via strongly-consistent locking is clearly a overkill when it comes to such a sparse access pattern. Furthermore, if multiple lock manager replicas are deployed for fault tolerance, strong locking pays the additional penalty of keeping the replicas consistent. On the other hand, the optimistic method of concurrency control enabled by Minuet can progress without the heavy locking overhead, while ensuring update atomicity in the rare cases of conflicting access. Although I/O requests are susceptible to rejection at the storage device, the rejection penalty does not appear to have a measurable effect on the overall application throughput.

The rate of I/O rejection could become more significant when a system has resource hotspots (e.g., index blocks in a database), but weakly-consistent locking can still provide a reasonable performance in such scenarios, since traditional strong locking would also face explosive synchronization overhead. Figure 4 shows the throughput and the rate of I/O rejection under a high- contention (*skewed(5/95)* workload). In this experiment, *strong(1)* displays higher throughput than *weak-own*, since rejected I/O requests, whose percentage goes up to 22%, have greater impact than denied lock requests, whose percent-

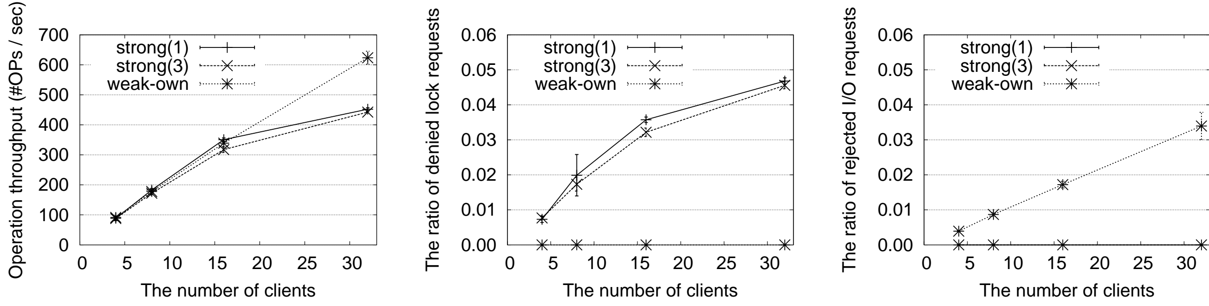

Figure 3: Throughput (left), the percentage of denied lock requests (center), and the percentage of rejected I/O requests (right) under the *uniform* workload.

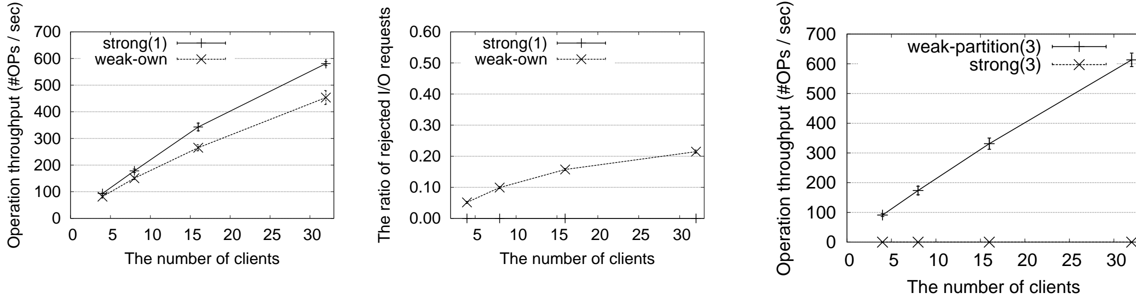

Figure 4: Throughput (left), and the percentage of rejected I/O requests (right) under the *skewed(5/95)* workload.

Figure 5: Throughput under the *uniform* workload in a network partition scenario.

age goes up to 38% (not shown). We also experimented varying the skewness of the workload and found that when the skewness decreases slightly (e.g., *skewed(10/90)*), the number of I/O rejections (and, accordingly, the performance difference) drops to a neglighble level.

Finally, Figure 5 shows how strong and weak locking protocols behave in a partitioned network scenario, where each client can communicate with only one lock manager replica out of three. A strongly-consistent locking protocol demands a well-connected primary component containing at least a majority of manager replicas - a condition that our partitioned scenario fails to satisfy. As a result, no client can make progress with traditional strong locking and the overall application throughput is zero. In contrast, under Minuet's weak locking (*weak-partition(3)*), clients can still make good progress. This experiment demonstrates the availability benefits that our approach gains over a traditional DLM design by loosening the consistency of locking state.

5.3 Distributed transactional update

The distributed transactional update application modifies multiple blocks, typically residing on different disks, in a single atomic transaction. As in previous experiments, we set the application block size to 4 KB.

Table 1 shows the throughput under *strong(1)* and *weak-own* modes of locking, varying the number of clients under the *uniform* workload. The two schemes do not demonstrate a measurable difference in throughput because the rate of contention (and, accordingly, transaction aborts) is relatively low. The trends of denied lock requests and rejected I/O requests for this experiment are similar to the ones observed for the simple distributed chunkmap.

scheme	no. of clients			
	4	8	16	32
strong(1)	79(2)	145(3)	208(13)	190(7)
weak-own	72(4)	123(7)	175(8)	203(17)

Table 1: Transaction throughput with the *uniform* workload, varying the number of clients.

6 Discussion

In this section, we discuss and address several concerns pertaining to the practical feasibility of our approach and the implications of Minuet's programming model.

Practical feasibility In this report, we explore a novel approach to concurrency control for SAN environments that rests on the basic idea of extending network-attached storage devices with a small amount of guard logic that enables them to detect and filter out inconsistent I/O requests. Fundamentally, this requires extending disk hardware with the guard functionality and modifying existing block-level I/O protocols to carry a certain amount of additional state (referred to as *capsules* in our design), which

may raise several concerns about the practical feasibility of our approach.

On the one hand, we acknowledge that our approach assumes the presence of functionality that does not exist in traditional disk hardware and, consequently, faces a non-trivial barrier to deployment. On the other hand, we observe that the proposed set of changes is very incremental in its nature and does not require extending storage devices with application-specific functionality. The guard logic presented in Algorithm 1 is amenable to efficient implementation in hardware, requiring only several table lookups and comparison operations.

As we tried to demonstrate in this report, the benefits of implementing such an extension can be substantial. In addition to lifting the safety and liveness limitations that have traditionally characterized applications and middleware in shared-disk environments, our approach establishes a new degree of freedom in the design space of SAN concurrency protocols, enabling a choice between optimism and strict coordination.

Different programming model Another concern is that Minuet introduces an alternative programming model, exposing application developers to several additional exception cases that do not typically arise under strongly-consistent locking. When a traditional DLM service grants a lock to an application process, the lock is assumed to be valid and hence, the process may proceed to accessing the respective shared resource without worrying about conflicting access from other processes. In contrast, Minuet gives out locks in a more permissive manner, but provides machinery for detecting and resolving inconsistent access at the storage device. As a result, applications that rely on Minuet for concurrency control must be programmed under the assumption that any I/O request to a remote disk may fail with *EBADSESSION* due to inconsistent locking state and take an appropriate corrective action (e.g., reacquire the lock and restart the operation).

We observe that while I/O request rejection does not occur under strongly-consistent locking, the protocols employed by traditional DLMs for ensuring system-wide consistency of locking state inevitably expose application developers to analogous exception cases. For example, a network connectivity problem causing some application node to lose connectivity with the primary component (e.g., a majority of lock managers) would typically cause that node to observe a DLM-related exception event. More specifically, the application process would be informed that the lock service is unreachable and, as a result, some (or all) of the locally-held locks may no longer be valid - these are precisely the semantics of the *ForcedDowngrade* notification in our design. Thus, both models demand exception-handling and recovery logic for dealing with a forced revocation of a lock.

In practice, the necessity of maintaining globally-consistent locking state drastically constrains the range of feasible recovery actions in such scenarios. Commonly, if a node experiences loss of connectivity to the DLM service, the only meaningful recovery action is inducing a shutdown (e.g., by panicking the kernel) with the expectation that the DLM will eventually detect the failure and reclaim the locks, thus permitting the rest of the system to make progress.

While such a strategy is certainly applicable in our model (and hence, existing applications can be deployed without modification), this technique would not observe the availability benefits enabled by our approach. With Minuet, a node that finds itself partitioned from the rest of the cluster need not immediately give up all of its locks and instead, can execute a more granular recovery action. For example, the affected node can switch to optimistic concurrency and continue accessing its resources without attempting to coordinate its session ID selection with the rest of the cluster and this would allow the partitioned node to continue making progress in the absence of conflicting access.

Our experience with developing and deploying sample applications (Section 4.4) on top of Minuet suggests that the availability benefits enabled by the use of fine-grained recovery actions are certainly worth the extra implementation effort, which we believe to be relatively small. The transactional chunkmap application was initially implemented on top of conventional locking using 460 lines of C code and extending the implementation to operate on top of Minuet required adding 43 lines of code to handle $ForcedDowngrade$ and $XactSvcFailure$ events.

Storage and bandwidth overhead In our prototype implementation, target storage devices maintain 16 bytes of per-resource metadata (*targSID* and *targCSID*). For a traditional middleware component such as a database or a filesystem, a resource would typically correspond to a single fixed-length block containing application data or metadata and taking a clustered filesystem as an example, blocksizes in the range 128KB - 1MB are considered common [25]. Assuming 128KB application blocksize, our design incurs a storage overhead of 0.01%.

Perhaps more alarmingly, the table of per-resource metadata, indexed by *resourceID*, must be stored in random-access memory for efficient lookup on the data path. We envision the use of flash memory or battery-backed RAM for this purpose and observe that today, high-performance disk systems make extensive use of NVRAM for asynchronous write caching [26, 27].

A request capsule carrying the tuple $\langle resourceID, sType, SID, curCSID, nextCSID \rangle$ adds 29 bytes of overhead to each *Read* and $Write$ request sent over the SAN. While non-trivial for very small I/O requests, we consider this overhead to be manageable under most workloads, especially since parallel applica-

tions are often configured with a large I/O request size to achieve efficiency. During the verification phase of a transaction, the application node sends out n distinct $Verify$ requests, each carrying a 29-byte capsule, where n is the total number of resources touched by the respective transaction.

Cache coherence semantics Another concern is that locking is commonly used as a mechanism for distributed cache coherence and that the loosely-consistent paradigm we explore here cannot easily support the *strict coherence* semantics, where a *Read* request must always return the results of the most recent $Write$.

We make the following observations: First, our study focuses on addressing the issues of concurrency control in a fully *asynchronous* distributed setting, where the notion of "most recent" may not be well-defined. Suppose, however, that there exists an external physical clock that allows us to order application's *Read* and $Write$ requests to the cache and let $C(H)$ denote such ordering of requests for a given execution history H. We note that in an asynchronous distributed system with failures, even a strongly-consistent DLM service cannot guarantee strict coherence consistent with $C(H)$. Furthermore, we conjecture that no mechanism providing such guarantees can exist in an error-prone asynchronous system for the same reason that strict mutual exclusion cannot be attained non-trivially in such a setting.

7 Related Work

Concurrency control has been extensively studied in the operating systems, distributed systems, and database communities. VMS [28] was among the first widely-available operating systems to provide application developers with the abstraction of a general-purpose distributed lock manager. Since then, DLMs have been widely adopted for various purposes and today, they are viewed as a useful general-purpose building block for distributed applications and middleware.

Clustered filesystems (GFS [1], OCFS [2], PanFS [3], GPFS [4], Lustre [5], Xsan [6]) and relational databases (Oracle RAC [7]) rely on a distributed lock manager to coordinate parallel access to application data, metadata, and logs residing on shared disks. OpenDLM [29] is a widely-adopted general-purpose DLM implementation for Linux, currently used by GFS [1] and other clustered filesystems.

In web service data centers, distributed locking services such as Chubby [30] and Zookeeper [31] have also become popular. These services are intended primarily for *coarse-grained* synchronization - a typical use case might be to elect a master among a set of Bigtable [32] servers. Although the intended use of Minuet is to provide *fine-grained* synchronization in a shared-disk cluster, our system can also support such use cases by transitioning to strongly-consistent locking, whereby each lock is acquired with a coordination factor of 1. However, the availability improvements enabled by our approach would not apply in such scenarios. Unlike our system, Chubby provides a hierarchical resource namespace and the ability to store small pieces of data, in effect offering a filesystem-like abstraction, but these features are largely orthogonal to our core approach and can be retrofitted onto the current design if needed. Chubby's *lock sequencer* mechanism allows servers to detect and discard inconsistent client requests submitted under the protection of an outdated lock and our timestamp-based *sessionID* generalizes this idea to support shared-exclusive locking. We also develop this notion further and observe that once we have the ability to detect and reject out-of-order requests at the destination, very little is gained by enforcing strong consistency on replicated lock management state and specifically, the use of an agreement protocol (e.g., Paxos [12]) may be an overkill.

Concurrency control and transaction mechanisms have been extensively studied in databases. ARIES [33] is a state-of-the-art transaction recovery algorithm for a centralized database, supporting fine-granularity locking and partial rollbacks of transactions, while D-ARIES [10] extends this work to be usable in distributed shared-disk databases. Implementing these mechanisms on top of Minuet's locking and I/O facilities would ensure that they retain their safety properties in the face of arbitrary asynchrony. Minuet's basic transaction service presented in Section 3.3 incorporates elements of write-ahead logging, timestamp ordering, and two-phase commit, all of which are standard and well-known techniques in database design. Finally, database researchers have explored hybrid approaches to concurrency control [34] that enable tradeoffs between optimism and strict coordination and our work enables similar tradeoffs for applications deployed in a SAN environment, where data resides on application-agnostic block storage devices.

8 Conclusion and Future Work

In this report, we investigate a novel approach to concurrency control and transaction recovery in storage area networks. Today, clustered SAN applications coordinate access to shared state on disk using strongly-consistent locking protocols, but they are subject to safety and liveness problems in the presence of asynchrony and failures. We argue that strict mutual exclusion is neither necessary nor sufficient for application-level correctness and that there are several advantages to loosening the consistency requirements found in traditional locking protocols.

We augment SAN target devices with a small amount of logic called a guard, which enables us to provide a property called session serializability and a relaxed model of locking. These, in turn, provide a foundational building

block for more complex and useful application semantics such as distributed transactions.

We have designed, implemented, and evaluated Minuet, a DLM-like synchronization primitive for SAN applications based on the techniques and protocols we presented. Our evaluation suggests that distributed applications built atop Minuet enjoy good performance and availability, while guaranteeing safety.

We are currently working on expanding the set of sample applications to include a distributed B-tree and there remains substantial work to be done in terms of understanding and evaluating the performance and availability tradeoffs enabled by our approach. The results we present in this report focus primarily on comparing the traditional strongly-consistent locking technique with a purely optimistic method enabled by Minuet, but these may be viewed as two opposites extremes of a continuum that invites further exploration. Finally, we plan to conduct a direct quantitative comparison between Minuet and a state-of-the art conventional lock service such as OpenDLM [29] and measure the differences in application availability and performance.

References

[1] Red Hat Global File System. http://www.redhat.com/gfs/.

[2] Oracle OCFS. http://oss.oracle.com/projects/ocfs/.

[3] Panasas PanFS. http://www.panasas.com/panfs.html.

[4] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[5] SUN Lustre. http://www.lustre.org/.

[6] Apple Xsan. http://www.apple.com/xsan/.

[7] Oracle Real Application Clusters. http://www.oracle.com/technology/products/database/clustering/index.html.

[8] Yuh-Jzer Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC*, pages 51–60, 1998.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[10] Jayson Speer and Markus Kirchberg. D-ARIES: A distributed version of the ARIES recovery algorithm. *ADBIS Research Communications*, 2005.

[11] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[12] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[13] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. In *PODC ’85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 163–174, New York, NY, USA, 1985. ACM.

[14] Gary L. Peterson. An O(nlog n) unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, 1982.

[15] D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, 1980.

[16] Remote system management using the Dell remote access card. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_bell.

[17] HP remote insight lights-out edition II (QuickSpecs). http://h18013.www1.hp.com/products/quickspecs/11377_div/11377_div.pdf.

[18] Brocade 5300 switch. http://www.brocade.com/products/switches/5300_fibre_channel_switch.jsp.

[19] SCSI-3 block commands (draft proposed standard). http://www.t10.org/ftp/t10/drafts/sbc/sbc-r08c.pdf.

[20] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD ’88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM.

[21] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system, 2003.

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[23] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[24] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *Readings in database systems (2nd ed.)*, pages 209–215, 1994.

[25] OCFS best practices. http://oss.oracle.com/projects/ocfs/dist/documentation/UL_best_practices.txt.

[26] Jai Menon and Jim Cortney. The architecture of a fault-tolerant cached RAID controller. *SIGARCH Comput. Archit. News*, 21(2):76–87, 1993.

[27] R. Treiber and J. Menon. Simulation study of cached RAID5 designs. In *HPCA ’95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 186, Washington, DC, USA, 1995. IEEE Computer Society.

[28] Jr. William E. Snaman and David W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, September 1987.

[29] OpenDLM. http://opendlm.sourceforge.net.

[30] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of OSDI*, 2006.

[31] ZooKeeper. http://sourceforge.net/projects/zookeeper.

[32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chan-

dra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of OSDI*, 2006.

[33] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

[34] Shirish H. Phatak and B. R. Badrinath. Bounded locking for optimistic concurrency control. *Rutgers University, Technical Report No. DCS-TR-380*, 1999.

A Proof of Claim 1

Proof. Suppose that the claim is false, which would mean that there exists an execution history H, in which session serializability on some shared resource R is violated. By the definition, R's owner device, which we denote $D(R)$, would observe in this history a sequence of I/O requests on R of the form $\langle ...r_i^s, ..., r_k^{s*}, ..., r_j^s, ...\rangle$, where r_i^s and r_j^s are a part of session s from some client c and r_k^{s*} is a part of session s^* from c^* that conflicts with s.

We first consider the case where s is *Shared* session, which means that s^* must be an *Exclusive* session and let $SID = \langle T_s, T_x\rangle$ and $SID^* = \langle T_s^*, T_x^*\rangle$ denote the corresponding session identifiers. The capsule evaluation logic at $D(R)$ would accept requests $\langle r_i^s, ...r_k^{s*}, ..., r_j^s\rangle$ in that order only if $T_x^* \geq T_x$ and $T_x \geq T_x^*$, which implies $T_x = T_x^*$. Furthermore, since s^* is an *Exclusive* session, we have $T_s^* \geq T_s$ and by uniqueness of session proposals, T_s^* is strictly greater than T_s. However, $T_x^* = T_x$, which means that T_x^* was reflected in c's maximum timestamp estimate (i.e., $MaxT_x(c,R) = T_x^*$) at the time of c's session proposal. This means that c must have previously (a) Made an unsuccessful attempt to acquire a lock with an exclusive timestamp T_x' smaller than T_x^* and received an $UpgradeDenied$ response from the lock manager. (b) Received an *EBADSESSION* response to an earlier I/O request on R with an outdated timestamp T_x' because the device has already accepted a request from c^* with $T_x^* > T_x'$.

In both cases, c would update its $MaxT_s$ and $MaxT_x$ estimates to reflect the values chosen by c^* and therefore, when c proposes a session identifier for request r_i^s, we have $MaxT_s(c,R) \geq T_s^*$. When establishing a *Shared* session, the protocol requires c to propose a shared timestamp greater than its current $MaxT_s$, which means that T_s is strictly greater than T_s^* - a contradiction.

An analogous argument, which we omit for brevity, demonstrates that if s is an *Exclusive* session then no two requests from s are interleaved at $D(R)$ by a conflicting request from s^*. □