Minuet: Rethinking Concurrency Control in Storage Area Networks

Andrey Ermolinskiy[†], Daekyeong Moon[†], Byung-Gon Chun[⋆], Scott Shenker[† ‡]
[†]*University of California at Berkeley*, [⋆]*Intel Research Berkeley*, [‡]*ICSI*

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. 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* via a *globally-consistent view of locking state* is neither sufficient nor strictly necessary to ensure application-level correctness in the presence of asynchrony and failures. We also argue that in many 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, crash 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 [19] and are proving effective in supporting a range of applications across a broad spectrum of industries. According to a recent survey of IT professionals across a range of corporations, government agencies, and universities, the overwhelming majority (80%) have deployed a storage area network in their organizations and 26% of the respondents report having deployed five or more SANs [14]. 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 SAN architecture is a particularly attractive choice for parallel clustered applications that demand high-speed concurrent access to a scalable storage backend. Such applications commonly rely on a clustered middleware service to provide a higher-level storage abstraction such as a filesystem (GFS [35], OCFS [8], PanFS [10], GPFS [37]) or a relational database (Oracle RAC [9]) on top of raw disk blocks.

One of the primary design challenges for clustered SAN applications and middleware 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). Typically, DML services aim to provide the guarantee of *strict mutual exclusion*, ensuring that no two processes in the system can simultaneously hold conflicting locks. In abstract terms, providing such guarantees requires enforcing a globally-consistent view of lock acquisition 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 close look at the semantics of SAN lock services and ask whether the assurances of full mutual exclusion and strongly-consistent locking are, in fact, a prerequisite for correct application behavior. Our main finding is that the standard semantics of mutual exclusion provided by a DLM are neither strictly necessary nor sufficient to guarantee safe coordination in the presence of node failures and asynchrony. In particular, processing and queuing delays in SAN switches and host bus adapters (HBAs) expose applications to out-of-order delivery of I/O requests from presumed faulty processes which, in certain scenarios, can incur catastrophic violations of safety and cause permanent data loss.

We propose and evaluate a new technique for disk access coordination in SAN environments. Our approach augments target storage devices with a tiny application-independent functional component, called a *guard*, and a small amount of state, which enable them to reject inconsistent I/O requests and provide a property called *session isolation*.

These extensions enable a novel *optimistic* approach to concurrency control in SANs and can also make existing lock-based protocols safe in the face of arbitrarily delayed message delivery, drifting clocks, crash process failures, and network partitions. The session isolation property in turn provides a foundational primitive for implementing more complex and useful coordination semantics, such as *serializable transactions*, and we demonstrate one such protocol.

We then describe the implementation of Minuet- a software library that provides a novel synchronization

primitive for SAN applications based on the protocols we present. Minuet assumes the presence of guard at the target storage devices and provides applications with locking and distributed transaction facilities, while guaranteeing liveness and data safety in the face of arbitrary asynchrony, node failures, and network partitions. Our evaluation shows that applications built atop Minuet compare favorably to those that rely on a conventional strongly-consistent DLM, offering comparable or better performance and improved availability.

Unlike existing services for fault-tolerant distributed coordination such as Chubby [20] and Zookeeper [15], Minuet requires its lock managers to maintain only loosely-consistent replicas of locking state and thus permits applications to make progress with less than a majority of lock manager replicas. To demonstrate the practical feasibility of our approach, we implemented two sample applications – a distributed chunkmap and a B+ tree – on top of Minuet and evaluated them in a clustered environment supported by an iSCSI-based SAN.

The benefits of optimistic concurrency control and the associated tradeoffs have been extensively explored in the database literature and are well understood. In particular, techniques such as callback locking, optimistic 2-phase locking, and adaptive callback locking [18, 21, 24, 42] have been proposed to enable safe coordination and efficient caching in client-server databases. It is important to note, however, that these approaches are not directly applicable to SANs because they assume the existence of a central lock server, typically co-located with the data block storage server. This assumption does not hold in a SAN environment, where the storage "servers" are application-agnostic disk arrays that possess no knowledge of locking state or node liveness status. Hence, a conservative DLM service that enforces strict mutual exclusion has traditionally been viewed as the only practical method of coordinating concurrent access to shared state for SAN applications.

Our main insight is that a single nearly trivial extension to the internal logic of a SAN storage device suffices to address the data safety problems associated with traditional DLMs and enables a very different approach to protocol layering for storage access coordination. Crucially, we achieve this without introducing application-level logic into storage devices and without forfeiting the generality and simplicity of the traditional block-level interface to SAN-attached devices.

The technical feasibility of device-based synchronization and its practical advantages have been demonstrated by several earlier proposals [12, 29]. Our study builds on this earlier work and while prior efforts have primarily focused on moving the functionality of a traditional cluster lock manager into the storage device, Minuet aims to provide a more general and useful synchronization primitive that supports a wider range of concurrency control mechanisms. In addition to supporting traditional *conservative* locking, our approach enables an *optimistic* method of concurrency control that can improve performance for certain application workloads. Further, Minuet allows existing locking protocols to remain safe in the presence of arbitrarily-delayed message delivery, node failures, and network partitions.

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

2 Background

2.1 Storage area networks

Storage area networks (SANs) are popular in enterprise datacenters and are commonly adopted to support the storage needs of data-intensive clustered applications. 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 local disks. Most SANs utilize a combination of SCSI and a low-level transport protocol such as TCP/IP or FCP (Fibre Channel Protocol) for communication between the application nodes and the target storage devices.

SANs aim to provide fully decentralized access to shared application state on disk and in principle, any SAN-attached client node can access any piece of data without routing its requests to a dedicated server. While in this model, all requests on a particular piece of data are centrally serialized, the crucial distinction from the traditional *server-attached* storage paradigm is that the point of serialization is a hardware disk controller that exposes an application-independent I/O interface on raw disk blocks and is oblivious to application semantics and data layout considerations.

Broadly, the SAN paradigm is advantageous from the standpoint of availability because it offers better redundancy and decouples node failures from loss of persistent state. Incoming application requests can be routed to any available node in the cluster and, in the event of a node failure, subsequent requests can be redirected to another processor with minimal interruption of service.

One of the primary design challenges for clustered SAN applications and middleware is ensuring safe and

efficient coordination of access to shared state on disk and commonly, a software service called a *distributed lock manager* (DLM) is employed to provide such coordination. A typical lock service such as OpenDLM [7] operates on *shared resources*, abstract application-level entities that require access coordination, and attempts to provide the guarantee of *mutual exclusion* - no two processes may simultaneously hold conflicting locks on the same resource.

2.2 Safety and liveness problems in SANs

In theory, DLM-based mutual exclusion offers sufficient mechanism to ensure safe access to shared state. In practice, however, guaranteeing safe serialization of disk requests tends to be more difficult than the above discussion might suggest due to the effects of *node failures* and *asynchrony*: nodes can fail by stopping and the processing and communication delays are not bounded. The following examples illustrate the nature of the problem.

Scenario 1: Consider a data structure S spanning 10 blocks on a shared disk D and two clients, C_1 and C_2, that are accessing the data structure concurrently. C_1 is updating blocks $[3-7]$ of S under the protection of an exclusive lock, while C_2 wants to read S in its entirety (i.e., blocks $[0-9]$) and is waiting for a shared lock. Suppose C_1 crashes after sending its $WRITE$ request to D but before hearing the response. The lock manager correctly detects the failure, reclaims the exclusive lock, and grants it to C_2 in shared mode. Next, C_2 proceeds to reading S and, assuming that a single disk request can carry up to 5 blocks of data, issues two requests: $R_1 = \langle READ[0-4] \rangle$ and $R_2 = \langle READ[5-9] \rangle$. Suppose C_1's delayed $WRITE$ request on blocks $[3-7]$ reaches the disk after R_1 but before R_2, in which case only the latter would reflect the effects of C_1's update. Hence, although individual I/O requests are processed by D as atomic units, their inconsistent interleaving would cause C_2 to observe and act upon a *partial update* from C_1, which can be viewed as a violation of data safety.

As an alternative to heartbeat failure detection, a lease-based mechanism [26] can be used to coordinate clients' accesses in the above example, but precisely the same problematic scenario would arise when clocks are not synchronized. When C_1 crashes and its lease expires, the lease manager could grant it to C_2 prior to the arrival of the last $WRITE$ from C_1 to the storage target. Since the target does not coordinate with the lease manager, it fails to establish the fact that an incoming request from C_1 is inconsistent with the current lease ownership state.

Scenario 2: Clustered applications and middleware services commonly need to enforce transactional semantics on updates to application state and metadata. In a shared-disk clustered environment, distributed transactions have traditionally been supported by two-phase locking in conjunction with a distributed write-ahead logging (WAL) protocol. In the abstract, the system maintains a snapshot of application state along with a set of per-client logs (also on shared disks) that record *Redo* and/or *Undo* information for every transaction along with its commit status. 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 flushed to the snapshot prior to the failure. An essential underlying assumption here is that once log recovery is initiated, no additional $WRITE$ requests from the suspected process will reach the snapshot. A violation of this assumption could result in the corruption of logs and application data.

Ensuring data safety in a shared-disk environment has traditionally required a set of *partial synchrony assumptions* to allow reliable heartbeat-driven failure detection and/or leases. For example, lease-based mechanisms typically expect bounded clock drift rates and message delivery delays to ensure the absence of in-flight I/O requests upon lease termination. However, these assumptions are probabilistic at best and since application data integrity is predicated on the validity of these assumptions, failure timeouts must be tuned to a very conservative value to account for worst-case delays in switch queues and client-side buffering. Such (necessarily) pessimistic timeouts may have a profoundly negative impact on failure recovery times - one of the common criticisms of SAN-oriented applications [16].

Another serious limitation exhibited by today's SAN applications is *liveness*. The DLM (or lease 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 a consistent order, which necessitates consensus mechanisms such as Paxos [31]. Alternatively, a single lock manager instance can be elected dynamically [27] from a group of candidates and in this case, ensuring mutual exclusion necessitates global agreement on the lock manager's identity. In both cases, reaching agreement fundamentally requires access to an active primary component - typically a majority of nodes. As a result, a large-scale node failure or a network partition that renders the primary component unavailable or unreachable may bring about a system-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) [3], SAN fabric fencing [1], and SCSI-3 PR [11] can be employed to mitigate some of these issues. These mechanisms help reduce the likelihood of data corruption under common failure scenarios, but do not provide the desired assurances of safety and liveness in the general case and, as we would argue, do not address the underlying problem. We observe that the underlying problem may be a case of *capability mismatch* between "intelligent" application processes that possess full knowledge of application's data structures, their disk layout, and consistency semantics on the one hand and relatively "dumb" storage devices on the other. The safety and liveness problems illustrated above can be attributed to a 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*.

3 Minuet Design

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 enforce the *mutual exclusion* property: no two clients 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 [23] in such an environment. Furthermore, as we explain in the previous section, a hypothetical lock service that does offer such guarantees would not by itself suffice to guarantee data safety in such a setting due to the possibility of out-of-order I/O request delivery.

Rather than restricting access to critical code sections, 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 alternate notion of correctness as *session isolation*.

We define this correctness property in formal terms below and then present a protocol that achieves session isolation with the help of guard logic. Finally, we demonstrate how distributed multi-resource transactions can be supported using session isolation as a building block.

3.1 Session isolation

Throughout this paper, we will use the term *resource* to denote the basic logical unit of concurrency control. Each resource R is identified by a unique and persistent application-level identifier (denoted $R.resID$) and has some physical representation on a SAN-attached storage device, which we call its *owner* ($R.owner$). More concretely, a resource may represent a filesystem block, a database table, or an individual tuple in a table. An application process operates on R by issuing $READ/WRITE$ commands to $R.owner$, as well as by acquiring and releasing locks on $R.resID$. We begin by defining the notion of *session* to a shared resource and describing the *session isolation* criterion.

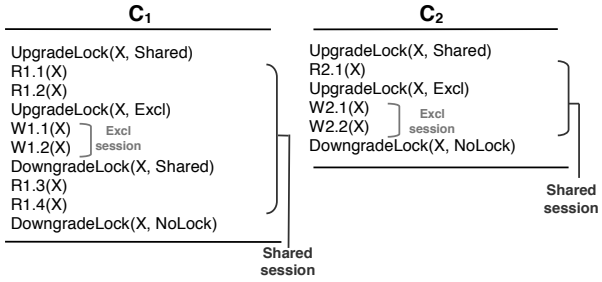

Figure 1: **Concurrent request streams to a shared resource X from two client processes, C_1 and C_2. In this example, C_1 first performs two $READ$ operations on X under the protection of a $Shared$ lock, then upgrades to $Excl$ and issues two $WRITE$s. Lastly, C_1 downgrades its lock to $Shared$ and 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.**

Definition 1. *If a client process C requests a Shared lock on R and the request is granted by the lock service, we say that C establishes a* ***shared session*** *to R. An existing shared session is terminated when C releases the Shared lock (i.e., downgrades to None). Analogously, by acquiring an Excl lock, a client establishes an* ***exclusive session*** *to R that can subsequently be terminated by downgrading to Shared or None.*

We define $Sessions(T,C,R)$ to be the set of all sessions to R from C active at time T, which is determined solely by the sequence of C's prior upgrade and downgrade requests to the lock service. $Sessions(T,C,R)$ may contain a shared or an exclusive session to R, or both, or none.

We say that a shared session ***conflicts*** *with every exclusive session to the same resource R and an exclusive session* ***conflicts*** *with every other session to R.*

Definition 2. *If a client process C issues at time T a disk request r that operates on shared resource R, we say that r* ***belongs to*** *session S if $S \in Sessions(T,C,R)$. For a given session S, we additionally define $Requests(S)$ to be the set of all disk requests that belong to S.*

Definition 3. *A given global execution history satisfies* ***session isolation*** *with respect to R if the sequence of disk request messages $M = \langle r_1, r_2, ... \rangle$ observed and processed in this history by R.owner satisfies: $\forall r_i, r_j \in M$ such that $\{r_i, r_j\} \subset Requests(S)$ for some $S : \nexists r_k \in M$ such that $i < k < j$ and $r_k \in Requests(S^*)$ for a session S^* from another client that conflicts with S.*

Informally, the above condition requires $R.owner$ to observe the prefixes of all sessions to R in strictly se-

rial order, ensuring that no two requests in a client's session are interleaved by a conflicting request from another client. To illustrate this definition, consider a pair of concurrent request sequences shown in Figure 1. In this scenario, the following two orderings of request observations by the owner of shared resource X would satisfy session isolation:

$$E_1 = \langle R_{1.1}, R_{1.2}, W_{1.1}, W_{1.2}, R_{1.3}, R_{1.4}, R_{2.1}, W_{2.1}, W_{2.2} \rangle$$
$$E_2 = \langle R_{1.1}, R_{1.2}, W_{1.1}, R_{2.1}, W_{2.1}, W_{2.2} \rangle$$

However, an execution history that causes the owner to observe $\langle R_{1.1}, R_{2.1}, R_{1.2}, W_{1.1}, W_{2.1} \rangle$ does not obey session isolation because it permits $R_{2.1}$ and $W_{2.1}$, two shared-session requests from C_2, to be interleaved by $W_{1.1}$, an exclusive-session request from C_1.

Note that session isolation 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 provided by session isolation are precisely those that applications developers have come to expect from a traditional DLM. To see this, observe that in the previous example, a conventional lock service offering full mutual exclusion would cause X to observe E_1 by granting clients' requests in the order $\langle C_1(Shared), C_1(Excl), C_2(Shared), C_2(Excl) \rangle$. Likewise, E_2 corresponds to a possible failure scenario in which C_1 crashes after acquiring its locks, causing the DLM to reclaim them and grant ownership to C_2.

3.2 Guard

Our core approach is inspired by earlier work on bridging the intelligence gap between applications and block storage devices [17, 25], as well as earlier proposals for device-based synchronization [12, 29]. We augment SAN-attached disks with a small application-independent component, which we call a *guard*, that enforces the session isolation invariant on the stream of incoming I/O commands. We associate a *session identifier* (*SID*) with every client session to a shared resource and modify the storage protocol stack on the initiators to annotate all outgoing disk commands with the current *SID* for the respective resource. Below, we refer to this additional state in the command header as *session annotation*.

A session annotation for a disk command operating on R has two components: a *session verifier* and a *session update*, denoted by $R.verifySID$ and $R.updateSID$, respectively. For commands that belong to an existing session, the verifier enables the target to confirm session validity prior to accepting the command and $updateSID$ is used by the initiator to signal the start of a new session.

For each shared resource R, its owner device maintains a local session identifier (denoted $R.ownerSID$) on persistent storage. Upon receipt of an I/O command from an initiator, the owner invokes the guard, which evaluates the command's session annotation against $R.ownerSID$ and determines whether session isolation would be preserved by accepting the command. Functionally, the guard operation is a form of compare-and-set and we describe this operation in detail in Section 3.3.

If an incoming I/O request fails verification, the target drops the request from its input queue and notifies the initiator via a special status code *EBADSESSION*. From an application developer's point of view, session rejection appears as a failed disk request along with an exception notification from the lock service indicating that a lock on the respective resource is no longer valid.

The guard situated at the 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 permits us to simplify the core functionality of the DLM module. In our scheme, the primary purpose of the lock service is ensuring an efficient assignment of session identifiers to clients that minimizes the frequency of command rejection for a given application workload.

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 every client selects its *SID*s via an independent local decision without attempting to coordinate with the rest 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 aims to position itself in the continuum between these extremes and allow application developers to trade off lock service availability, synchronization overhead, and I/O performance.

3.3 Enforcing session isolation

Minuet uses a simple timestamp-based mechanism to guarantee the session isolation invariant. 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. These timestamps are globally unique - no two sessions from distinct clients are identified using the same pair of values and no client is assigned the same value pair twice. To ensure global uniqueness, we use the following timestamp format: $\langle T.incNum.clntID \rangle$, where $clntID$ uniquely identifies the client process and $incNum$ is the client's *incarnation number* - a monotonic counter ensuring uniqueness across crashes.

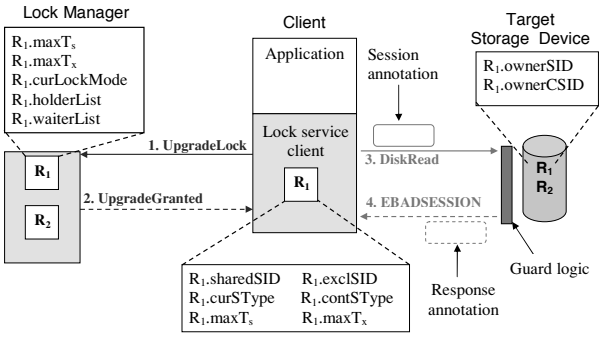

Figure 2: **Protocol messages and per-resource state at application clients, lock managers, and shared storage devices.**

Client-side state: For each shared resource R, a client C maintains a pair of session identifiers for its shared and exclusive sessions to R, denoted by $R.sharedSID$ and $R.exclSID$, respectively. Additionally, $R.curSType$ identifies the *current session type*, one of $\{None, Shared, Excl\}$, and $R.contSType$ holds the client's *session continuation type*. The latter value is used by the target device to verify (prior to executing a request from C) that its existing session has not been broken by a conflicting request from another client. Finally, every client C maintains an estimate of the largest shared and exclusive timestamp values previously assigned to a session identifier for any client, which we denote by $R.maxT_s$ and $R.maxT_x$. Initially, $R.sharedSID = R.exclSID = NIL$, $R.curSType = R.contSType = None$, and $R.maxT_s = R.maxT_x = 0$. The steps and states of the basic locking and storage access protocols are illustrated in Figure 2.

Acquiring locks: To acquire/upgrade a lock on resource R, a client C *proposes* a unique session timestamp pair $\langle proposedT_s, proposedT_x \rangle$ to the lock manager. To acquire a *Shared* lock on R, C sets $proposedT_x \leftarrow R.maxT_x$ and sets $proposedT_s$ to some unique timestamp greater than $R.maxT_s$. The 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 been accepted. Otherwise, the manager denies the request and responds with $UpgradeDenied$, which includes the largest timestamp values observed by the manager. In the latter case, the client updates its local estimates $\langle R.maxT_s, R.maxT_x \rangle$ and submits a new proposal. After accepting and enqueuing C's request, the lock manager eventually grants it and responds with $UpgradeGranted$. The client then sets $R.curSType \leftarrow Shared$ and initializes the shared session identifier: $R.sharedSID \leftarrow \langle proposedT_s, proposedT_x \rangle$.

To upgrade a lock from *Shared* to *Excl*, the client sends $UpgradeLock$ to the lock manager after setting $proposedT_s \leftarrow R.maxT_s$ and $proposedT_x$ to some unique timestamp greater than $R.maxT_x$. Upon receiving $UpgradeGranted$ from the lock manager, the client sets $R.curSType \leftarrow Excl$ and $R.exclSID \leftarrow \langle proposedT_s, proposedT_x \rangle$. Upgrading from $None$ to $Excl$ is functionally equivalent to acquiring a *Shared* lock and then upgrading to $Excl$, but as an optimization, these operations can be combined into a single request to the lock manager.

Setting up a session annotation during I/O request submission:

```
if (R.curSType = Shared)          /* Shared session is active */
  R.updateSID ← R.sharedSID;
  R.verifySID.Ts ← NIL;  R.verifySID.Tx ← R.sharedSID.Tx;
else                              /* Exclusive session is active */
  R.updateSID ← R.exclSID;
  if (R.contSType = Shared)
    R.verifySID.Ts ← NIL;  R.verifySID.Tx ← R.sharedSID.Tx;
  else
    R.verifySID ← R.exclSID;
```

Guard logic at the target device:

```
Use the resource identifier (R.resID) to look up R.ownerSID;
if (R.verifySID.Tx < R.ownerSID.Tx) REJECT;
if (R.verifySID.Ts ≠ NIL)
  if (R.verifySID.Ts < R.ownerSID.Ts) REJECT;
if (REJECTED)
  Respond to the initiator with ⟨EBADSESSION, R.ownerSID⟩;
else
  R.ownerSID.Ts ← Max(R.ownerSID.Ts, R.updateSID.Ts);
  R.ownerSID.Tx ← Max(R.ownerSID.Tx, R.updateSID.Tx);
  Execute the command and respond to the initiator;
```

Figure 3: **Disk request submission and guard logic pseudocode.**

Accessing shared storage: After establishing a session to R by acquiring a corresponding lock, client can proceed to issuing disk requests that operate on the content of R. Each outgoing request is augmented with a session annotation that enables the target device to verify proper ordering of requests and enforce session isolation. The annotation carries a tuple of the form $\langle R.resID, R.verifySID, R.updateSID \rangle$ and is initialized as shown in Figure 3.

Upon receipt of a disk request from a client, the owner device invokes the guard logic, which evaluates the session annotation as specified in Figure 3. In the event of rejection, the owner immediately discards the command and sends an *EBADSESSION* response to the client, together with a *response annotation* carrying $\langle R.ownerSID \rangle$. Otherwise, the owner executes (or enqueues) the command and updates its local session identifier as shown in the figure.

Upon receipt of an *EBADSESSION* status code, the initiator examines the response annotation and notifies the application process that its lock and session on R is no longer valid. The condition $R.verifySID.T_s < R.ownerSID.T_s$ indicates interruption of an exclusive session, in which case the client downgrades its lock to *Shared*, sets $\langle R.curSType, R.contSType \rangle \leftarrow Shared$, and sets $R.exclSID \leftarrow NIL$. A *Shared* lock is further downgraded to $None$ if $R.verifySID.T_x < R.ownerSID.T_x$

(since in this case, a conflicting exclusive-session request has been accepted). In this situation, the client sets $\langle R.curSType, R.contSType \rangle \leftarrow None$ and $R.sharedSID \leftarrow NIL$. In both cases, the maximum timestamp estimates $R.maxT_s$ and $R.maxT_x$ are updated to reflect the most recent timestamps observed by the owner.

Upon receiving a *SUCCESS* status code, the client sets $R.contSType \leftarrow R.curSType$ and updates the shared session identifier to reflect the most recent value in the annotation: $R.sharedSID \leftarrow R.updateSID$. (This step is necessary to ensure that a shared session remains valid after a $Shared \rightarrow Excl$ upgrade or a downgrade to *Shared*).

Downgrading locks: To downgrade an existing lock from *Excl* to *Shared*, the client sends a *DowngradeLock* request to the lock manager and resets the exclusive-session state: $R.exclSID \leftarrow NIL$, $\langle R.curSType, R.contSType \rangle \leftarrow Shared$. Similarly, to downgrade from *Shared* to *None*, the client notifies the lock manager and sets $R.sharedSID \leftarrow NIL$, $\langle R.curSType, R.contSType \rangle \leftarrow None$. Upon receipt of a *DowngradeLock* request, the manager updates the ownership state for R and, if possible, grants the lock to the next waiter in the queue.

Correctness: The locking protocol and the guard described above guarantee session isolation and a formal correctness argument can be found in [22]. Informally, consider two clients C_1 and C_2 that compete for shared and exclusive access to R, respectively, and suppose that a shared-session request from C_1 gets accepted with $R.updateSID = \langle T_s^1, T_x^1 \rangle$ in its annotation. Observe that due to global uniqueness of session proposals, the owner of R would subsequently accept an exclusive-session request from C_2 with verifier $R.verifySID = \langle T_s^2, T_x^2 \rangle$ *only if* T_x^2 is strictly greater than T_x^1. In this case, subsequent shared-session requests from C_1 would fail verification, causing C_1 to observe *EBADSESSION* and downgrade its lock. Thus, session isolation would be preserved in this example via a forced termination of C_1's session. A similar argument demonstrates that no two exclusive-session commands can be interleaved by a conflicting command from another client.

3.4 Supporting distributed transactions

3.4.1 Overview and design requirements

Transactions are widely regarded as a useful programming primitive and traditionally, SAN-oriented applications implement transactional semantics using two-phase locking for isolation and a write-ahead logging (WAL) facility (sometimes referred to as *journaling*) for atomicity and durability. To support transactions, Minuet relies on these well-understood and widely-used mechanisms, while extending them with the use of the guard to address the safety problems outlined in Section 2.2. Since the primary focus of this paper is feasibility of safe and highly-available applications in SANs rather than performance, we provide only a subset of features typically found in a state-of-the-art transaction service such as D-ARIES [38]. Below, we present a design that implements *Redo*-only logging to support the "no force no steal" buffer policy and currently, our design permits only one active transaction per process at a time - after starting a transaction, a client must commit or abort before initiating 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 easily retrofitted onto our scheme if necessary. The following set of requirements motivates our design:

(1) Avoid introducing assumptions of synchrony required by conventional transaction schemes for SAN environments. We rely on the guard at target devices to provide session isolation and protect the state on disk from the effects of arbitrarily-delayed I/O commands operating on the application data and the log.

(2) Eliminate reliance on strongly-consistent locking. Rather than requiring clients to coordinate concurrent activity via a strongly-consistent DLM, the guard at storage devices enables a limited form of isolation and permits us to relax the degree of consistency required from the lock service. Prior to committing a transaction, a client process in Minuet issues an extra disk request, which verifies the validity of all locks acquired at the start of the transaction. This mechanism allows us to identify and resolve cases of conflicting access due to inconsistent locking state at commit time and can be viewed as a variant of optimistic concurrency control - a well-known technique from the DBMS literature [30].

(3) Avoid enforcing a globally-consistent view of process liveness. Rather than relying on a group membership service to detect client failures and initiate log recovery proactively in response to perceived failures, our design explores a *lazy* approach to transaction recovery that postpones the recovery action until the affected data is accessed. This enables Minuet to operate without global agreement on group membership.

3.4.2 Basic transaction protocol

Minuet stores transaction *Redo* information in a set of per-client logs on shared disks. The physical location of a client's log can be computed from its client identifier (*clntID*). These logs appear to Minuet's transaction module as regular lockable resources that can be read from and written to, while the guard is assumed to enforce session isolation in the event of concurrent access from multiple clients.

To support transactions, we extend the basic session

isolation machinery described in Section 3.3 with an additional piece of state called a *commit session identifier (CSID)* of the form $\langle clntID, xactID \rangle$. We extend the format of a session annotation to include two commit session identifiers, denoted $verifyCSID$ and $updateCSID$, and both are set to NIL unless specified otherwise. For each shared resource R, the owner device maintains a local commit session identifier ($R.ownerCSID$) as well as $R.ownerSID$. Conceptually, the value of $R.ownerCSID$ at a particular point in time identifies the most recent transaction that may have updated R and committed without flushing its changes to the disk image of R. If $R.ownerCSID \neq NIL$, the current state of R on disk may be missing updates from a committed transaction and thus cannot be assumed valid. In this case, $R.ownerCSID.clntID$ identifies the client process responsible for the latest transaction on R and it is used to locate the corresponding log for recovery purposes.

Upon receiving a disk request, the guard examines the annotation and rejects the request if $R.verifyCSID.clntID \neq R.ownerCSID.clntID$ or if $R.verifyCSID.xactID < R.ownerCSID.xactID$. A request is accepted *only if* its $verifySID$ and $verifyCSID$ both pass verification and upon completing the request, the owner device updates its local commit session identifier by setting $R.ownerCSID \leftarrow R.updateCSID$. If verification fails, the owner responds with $EBADSESSION$ and attaches the tuple $\langle R.ownerSID, R.ownerCSID \rangle$ in a response annotation.

In Minuet, transactions proceed in five stages: *Begin*, *Read*, *Update*, *Prepare*, and *Commit* and we illustrate them using high-level pseudocode in [22]. During one-time client initialization, Minuet's transaction service locks the local client's log in $Excl$ mode. To begin a new transaction T, the client selects a new transaction identifier ($curXactID$) via a monotonic local counter and appends a *BeginXact* record to its log. Next, in the *Read* phase of a transaction, the application process acquires a *Shared* lock on every resource in $T.readSet$ and reads the corresponding data from shared disks into local memory buffers. In the *Update* phase that follows, the client acquires $Excl$ locks on the elements of $T.writeSet$, applies the desired set of updates locally, and communicates a description of updates to Minuet's transaction service, which appends the corresponding set of $Update$ records to the log. Each such record describes an atomic mutation on some resource in $T.writeSet$ and essentially stores the parameters of a single disk $WRITE$ command.

The *Prepare* phase serves a dual purpose: to verify the validity of client's sessions (and hence, the accuracy of cached data) and to lock the elements of the write set in preparation for committing. For each resource in $T.readSet \cup T.writeSet$, the client sends a special $PREPARE$ request to its owner. Minuet implements $PREPARE$ requests as zero-length $READ$s, whose sole purpose is to transport an annotation and invoke the guard. $PREPARE$ requests for elements of $T.writeSet$ carry $verifyCSID = NIL$ and $updateCSID = \langle C, curXactID \rangle$ in their annotations, where C is the client's identifier. If all $PREPARE$ requests return $SUCCESS$, the transaction enters the final *Commit* phase, in which a *CommitXact* record is force-appended to client C's log.

The protocol outlined above ensures transaction isolation, identifying cases of conflicting access during the *Prepare* phase. Recall, however, that under the session isolation semantics, any I/O command, including operations on the log, may fail with $EBADSESSION$ due to conflicting access from other clients. This gives rise to several exception cases at various stages of transaction execution. For example, a client C may receive an error while forcing *CommitXact* to disk due to loss of session to the log. This can happen only if another process has initiated log recovery on C and hence, the active transaction must be aborted. Other failure cases and the corresponding recovery logic are described in the report [22].

Syncing updates to disk: After committing a transaction, a client C flushes its locally-buffered updates to R simply by issuing the corresponding sequence of $WRITE$ commands to its owner device. Each such command specifies in its annotation $\{R.verifyCSID, R.updateCSID\} = \langle C, syncXactID \rangle$, where $syncXactID$ denotes C's most recent committed transaction that modified R. After flushing all committed updates, C issues an additional zero-length $WRITE$ request, which specifies $R.verifyCSID = \langle C, syncXactID \rangle$ and $R.updateCSID = NIL$ in the annotation. This request causes the device to reset $R.ownerCSID$ to NIL, effectively marking the disk image of R as "clean". Lastly, C appends to its log an $UpdateSynced$ record of the form $\langle R, syncXactID \rangle$.

Lazy transaction recovery: A client C can initiate transaction recovery when its disk command on some resource R fails with $EBADSESSION$ and a non-NIL value $ownerCSID = \langle C_F, xactID \rangle$ is specified in the response annotation. This response indicates that the disk image of R may be missing updates from a transaction committed earlier by another client C_F. If C suspects that C_F has failed, it invokes a local recovery procedure that tries to repair the disk image of R. First, C acquires exclusive locks on R and $C_F.Log$ and reads the log from disk. Next, C searches the log for the most recent transaction that has successfully flushed its updates to R, from which it determines the list of subsequent committed updates that may be missing from the disk image. The client then proceeds to repairing the state of R on disk by reapplying these updates and all $WRITE$ requests sent to the owner during this phase specify

$\{R.verifyCSID, R.updateCSID\} = R.ownerCSID$ in the annotation. Finally, after reapplying all missing updates, C completes recovery by issuing a zero-length $WRITE$ annotated with $R.verifyCSID = R.ownerCSID$, $R.updateCSID = NIL$. A more detailed discussion of transaction recovery in Minuet can be found in [22].

3.5 Lock manager replication

Some lock services seek to achieve fault tolerance by replicating lock managers. Since Minuet does not need to provide assurances of mutual exclusion, it relies on a simpler and more available replication scheme that permits clients to retain progress in the face of extensive node and connectivity failures. A lock can be acquired as long as at least one manager instance is reachable. In an extreme case, that instance can be the local Minuet client itself, which would simply grant its own proposals without coordinating with other processes.

To support manager replication, we extend the basic locking protocol presented in Section 3.3 as follows: When acquiring or upgrading a lock, a client selects a subset of managers, which we call its *voter set*, and sends an $UpgradeLock$ request to all members of this set. The lock is considered granted once $UpgradeGranted$ votes are collected from all members. If any of the voters respond with $UpgradeDenied$ due to an outdated timestamp, the client downgrades the lock on all members that have responded with $UpgradeGranted$, updates its $maxT_s$ and $maxT_x$ values, and resubmits the upgrade request with a new timestamp proposal. As a performance optimization, we allow $UpgradeLock$ requests to specify an *implicit downgrade* for an earlier timestamp.

4 Implementation

We have implemented a proof-of-concept prototype of Minuet based on the design presented in the preceding section. The prototype has been implemented on the Linux platform using C/C++ and consists of a client-side library, a lock manager process, an iSCSI protocol stack extension, and two sample parallel applications.

4.1 Core Minuet modules

Client-side library (5,440 LoC): The client-side component is implemented as a statically-linked library and provides an event-driven interface to Minuet's core services, which include locking, remote disk I/O, and transaction execution. When requesting a lock, a client can optionally specify the desired size of the voter set, which enables application developers to tune the degree of locking consistency, enabling a choice between optimism and strict coordination. A small voter set works well for low-contention resources; it helps keep the lock message overhead low and permits clients to make progress in a partitioned network. Conversely, a large voter set requires connectivity to more manager replicas, but reduces the rate of I/O rejection under high contention. All outgoing disk commands are augmented with session annotations and in the event of rejection by the target device, a *ForcedDowngrade* event is posted to inform the application that the corresponding lock has been downgraded to some weaker mode.

Minuet lock manager (4,285 LoC): The lock manager process grants and revokes locks using the timestamp mechanism of Section 3.3 and several manager replicas can be deployed for fault tolerance. For each lockable resource, the manager maintains the current lock mode, the list of current holders, the queue of blocked upgrade requests, and the largest observed timestamp proposal.

SAN protocols and guard logic: To demonstrate the practicality of our approach, we implemented the guard logic and session annotations within the framework of iSCSI [4], a widely-used transport for IP-based SANs, and our prototype extends an existing software-based implementation of the iSCSI standard. On application client nodes, we modified the top and the bottom levels of the 3-tier Linux SCSI driver model. The top-level driver (*/drivers/scsi/sd.c*) presents the abstraction of a generic block device to the kernel and converts incoming block requests into SCSI commands. We extended *sd* with a new *ioctl* call, which enables the Minuet client library to specify session annotations for outgoing requests and to collect response annotations.

The bottom-level driver implements a TCP encapsulation of SCSI and our current prototype builds on the Open-iSCSI Initiator driver [6] v2.0-869.2. We used the *additional header segment* (AHS) feature of iSCSI to attach Minuet annotations to command PDUs and defined a new AHS type for this purpose.

Our storage backend is based on the iSCSI Enterprise Target driver [5] v0.4.16, which exposes a local block device to remote initiators via iSCSI. We extended it with the guard logic, which examines incoming PDUs and makes an accept/reject decision based on the annotation. Command rejection is signaled to the initiator via the REJECT PDU defined by the iSCSI standard.

The addition of guard logic represents the most substantial extension to the SAN protocol stack, but incurs only a modest increase in the overall complexity. The initial implementation of the Enterprise Target driver contained 14,341 lines of code and augmenting it with Minuet guard logic required adding 348 lines.

4.2 Sample applications

Distributed chunkmap (342 LoC): 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 mutations to a distributed chunkmap - a common scenario in clustered middleware such as filesystems and databases. The chunkmap could represent a bitmap of free disk blocks, an array of i-node structures, or an array of directory file slots. In each iteration, the application selects a random chunk, reads it from shared disk, modifies a random chunk region, and writes it back to disk. To ensure update atomicity, the application acquires an *Excl* lock on the respective block from Minuet prior to reading it from disk and releases the lock after writing back the updated version.

Distributed B-Tree (3,345 LoC): To demonstrate the feasibility of serializable transactions, we implemented a distributed B-link tree [32] (a variant of B+ tree) on top of Minuet. Our implementation provides transactional *insert*, *delete*, *update*, and *search* operations based on the protocol presented in Section 3.4.2. For each operation, the application initiates a transaction and fetches the chain of tree blocks necessary for the operation (*Read* phase). Next, it upgrades the locks on the modified blocks to *Excl* mode and logs the updates (*Update* phase). Lastly, the client *Prepares* and *Commits* the transaction. If a transaction aborts due to loss of session to a tree block or the client's log, the application reacquires the corresponding lock and retries (without backoff) until it commits successfully. For efficiency, clients retain *Shared* locks (and the content of cache buffers) across transactions and stale cache entries are detected and invalidated during the *Prepare* phase.

5 Evaluation

In this section, we evaluate the performance of our applications under different modes of locking. Due to space constraints, we present only key results that demonstrate the benefits of optimistic coordination enabled by Minuet and confirm the feasibility of our design. Several additional important measurements are reported in [22].

5.1 Experimental setup

For our experiments, we emulated a 39-node SAN environment interconnected via 100Mbps links using Emulab [41] and detailed hardware specifications are provided in Figure 4. Three of the nodes were configured to serve as Minuet lock managers and four additional nodes were used to emulate SAN-attached target devices, collectively providing 2GB of logical disk space, equally striped across the nodes. The remaining 32 nodes were configured as application clients. We ran each iteration of the experiment for 5 minutes and all of the values reported below are averages over 3 iterations.

In each iteration, we measure the aggregate *goodput* (the number of successful application-level operations per second) from all nodes and the rate of disk command rejection under the following locking configurations:

***strong(x)*:** We deploy a total of $2x-1$ lock managers and require clients to obtain permissions from a majority (x). Note that *strong(1)* represents a traditional locking protocol with a single central lock manager, while *strong(2)* requires 3 lock manager replicas and masks one failure.

***weak-own*:** An extreme form of weakly-consistent locking. Each client obtains permissions only from the local lock manager (co-located on the same machine) and does not attempt to coordinate with the other clients.

In all of our experiments, applications rely on Minuet to provide both modes of locking and do not make use of any other synchronization facilities.

The number of storage targets	1	2	3	4
strong(1) coordination	105.0	220.0	329.9	412.4
strong(2) coordination	105.5	219.5	330.7	411.7
weak-own coordination	105.9	220.9	331.1	410.6

Table 1: **Chunkmap application goodput (in operations per second) under the *uniform* workload.**

5.2 Distributed chunkmap

In this experiment, we configured the chunk size to 8KB (for a total of 250K chunks) and ran the chunkmap application with 32 clients, varying the number of storage targets from 1 to 4. We considered two forms of workload:

***uniform*:** In each operation, a chunk to be modified is selected uniformly at random.

***hotspot(x)*:** x% of operations touch a *hotspot* region of the chunkmap constituting 0.1% of the entire dataset.

Table 1 reports the aggregate goodput under the *uniform* workload, which represents a low-contention scenario. The goodput exhibits linear scaling with the number of storage servers. Further, there is no measurable difference in performance between the three locking configurations. These results suggest that the optimistic method of coordination enabled by Minuet does not adversely affect application performance, while providing safety, in scenarios where the overall I/O load is high, but contention for a single resource is relatively rare.

The rate of I/O rejection increases when the workload has hotspots and, as expected, *weak-own* suffers a performance hit proportional to the popularity of the hotspot (Figure 5). We note that the *hotspot* workload represents a very stressful case (the hotspot size is 0.1%) and our results demonstrate that weakly-consistent locking degrades *gracefully* and can still provide reasonable performance in such scenarios.

We also ran experiments in a partitioned network scenario, where each client can communicate with only a subset of replicas. A strongly-consistent locking protocol demands a well-connected primary component containing at least a majority of manager replicas - a condi-

	Storage targets	Lock managers	Clients
# Nodes	4	3	32
CPU	3GHz Xeon	850Mhz Pentium III	
RAM	2GB	512MB	
Disk	10K RPM SCSI	7200 RPM IDE	

Figure 4: Hardware specifications of the Emulab cluster.

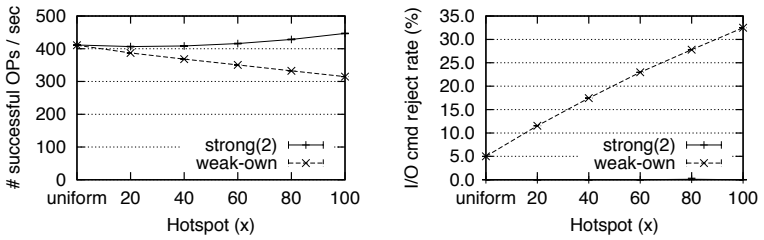

Figure 5: Left: chunkmap goodput under the *hotspot(x)* workload for varying x. Right: the rate of rejected I/O requests under *hotspot(x)* for varying x.

	tree-small	*tree-large*
Node size	8KB	8KB
Fanout	150	150
Num. keys	187,500	18,750,000
Leaf occupancy	50%	50%
Tree depth	3	4
Total dataset size	20MB	2GB

Figure 6: Pre-populated B+ tree parameters.

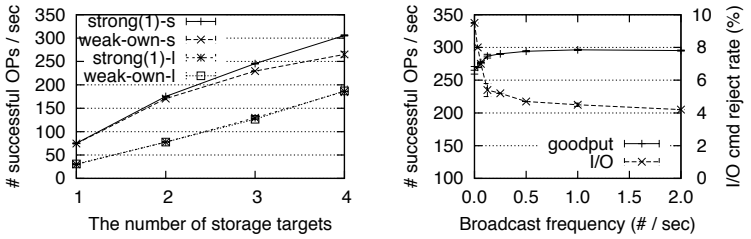

Figure 7: Left: B+ tree application goodput with *tree-small* and *tree-large* datasets. Right: effects of the timestamp broadcast optimization with *tree-small* dataset.

tion that our partitioned scenario fails to satisfy. As a result, no client can make progress with traditional strong locking and the overall application goodput is zero. In contrast, under Minuet's weak locking, clients can still make good progress. This demonstrates the availability benefits that Minuet gains over a traditional DLM design.

5.3 Distributed B+ tree

The B+ tree application demonstrates Minuet's support for serializible transactions. In this experiment, we start with a pre-populated tree and run the application for 5 minutes on 32 client nodes. Each client inserts a series of randomly-generated keys and we measure the aggregate goodput, defined as the total rate of successful insertions per second from all clients. To test Minuet's behavior under different transaction complexity and contention scenarios, we used two different pre-populated B+ trees, whose parameters are given in Figure 6.

Figure 7(left) compares the performance of *strong(1)* and *weak-own*. Under both locking schemes, the throughput exhibits near-linear scaling with the number of storage targets. As expected, *tree-large* demonstrates a lower aggregate transaction rate because each transaction requires accessing a longer chain of nodes. Moreover, since the number of leaf nodes is large, read-write or write-write contention is relatively infrequent and hence, the performance penalty due to I/O rejection incurred under *weak-own* is negligible. By contrast, *tree-small* represents a high-contention workload and our results suggest that even in this stressful scenario, Minuet's weak locking incurs only a modest performance penalty.

Further investigation revealed that the primary cause of the performance degradation was an outdated estimate of maximum timestamps $\langle maxT_s, maxT_x \rangle$, causing some of the commands to carry outdated session identifiers (e.g., with $verifySID.T_x < ownerSID.T_x$). Under *weak-own*, clients select session identifiers without coordinating with other clients and hence, a client may not know the up-to-date value of $ownerSID.T_x$ that may have been set by an earlier transaction from another client.

A simple optimization alleviating this issue is to let clients lazily synchronize their knowledge of maximum timestamps. More specifically, each client can broadcast its local updates on $maxT_s$ and $maxT_x$ to other clients at some fixed broadcast rate (b) and other clients can update their local estimates accordingly. We implemented this optimization and measured its effects on the *tree-small* workload with 4 storage targets. Figure 7(right) shows the results, which suggest that we can substantially reduce the rate of rejection by broadcasting with $b \geq 0.2$ and the resulting goodput closely approaches the maximum value achievable under strong locking.

Note that this optimization affects only performance and is not required for safety. Conceptually, the broadcast rate b provides a way of parameterizing the continuum between traditional locking and the fully optimistic case of *weak-own* and other methods may be possible.

6 Discussion

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

Storage target modifications: Our approach rests on the basic idea of extending SAN-attached storage targets with a small amount of guard logic that enables them to detect and filter out inconsistent I/O requests , which will require storage array vendors to introduce a new feature into their products.

We acknowledge that Minuet relies on functionality that does not presently exist in standard storage hardware and, consequently, faces non-trivial barriers to standardization, implementation, and deployment. However, we observe that the proposed extensions are very incremental and can easily be retrofitted into an existing design. The guard logic is amenable to efficient implementation in hardware or firmware, requiring only a few table lookups and comparison operations.

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

Our investigation builds upon earlier work on device locking [12], which has demonstrated the practical feasibility of this approach and the willingness of storage hardware vendors to adopt a promising new feature [2].

Metadata storage overhead: In our prototype implementation, target storage devices maintain 16 bytes of per-resource metadata. For a typical middleware service such as a database or a filesystem, a resource would correspond to a single fixed-length block containing application data or metadata and taking a clustered filesystem as an example, block sizes in the range 128KB - 1MB are common [8]. Assuming 128KB application block size, the table of Minuet session identifiers for a dataset of size 1TB would consume an additional 128MB.

Perhaps more alarmingly, Minuet metadata 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 storage arrays make extensive use of NVRAM for asynchronous write caching [39]. Alternatively, the session state can be stored persistently on disk and a fixed-size NVRAM buffer can be used as a cache, providing efficient access to the working set.

Protocol extensions: Our approach requires augmenting the format of *READ* and *WRITE* commands with session annotations and our prototype implementation extends the iSCSI protocol with a new AHS type for this purpose. A transport-level modification simplified our software implementation, but would be difficult to deploy in a production environment, since it would require modifying the HBAs. For a more easily-deployable solution, the required set of extensions can be implemented in a transport-independent manner at the SCSI command level. One option would be to use an *extended command descriptor block (XCDB)*, as defined in SPC-4 ([13], section 4.3.4), and introduce a new descriptor extension type for carrying the session annotation. Likewise, command rejection can be signaled to the initiator via a *CHECK CONDITION* status code with a new *additional sense code* and the response annotation can be communicated as *fixed-format sense data* ([13], section 4.5.3).

Programming model: Another concern is that Minuet imposes a different programming model, exposing application developers to additional exception cases that do not naturally arise under strong locking. When a traditional DLM service grants a lock to an application process, the lock is assumed to be valid and the client can proceed to accessing the disk without worrying about conflicting access from other clients. In contrast, Minuet gives out locks in a more permissive manner, but provides machinery for detecting and resolving conflicting access at the storage device. As a result, applications that rely on Minuet for concurrency control must be programmed with the assumption that any I/O request can fail with *EBADSESSION* due to inconsistent lock state.

We observe that while I/O rejection does not occur under conservative locking, the protocols employed by traditional DLMs for ensuring system-wide consistency of locking state inevitably expose application developers to analogous exception cases. For instance, a network connectivity problem causing some application node to lose connectivity to a majority of lock managers would typically cause that node to observe a DLM-related exception event. More concretely, the application process would be informed that due to lack of connectivity, some of its locks may no longer be valid - these are precisely the semantics of Minuet's *ForcedDowngrade* notification. Hence, both models demand exception-handling for dealing with forced lock revocation.

With Minuet, a node that finds itself partitioned from the rest of the cluster need not immediately give up its locks and instead, can perform a more granular recovery action. For example, it can switch to the optimistic method and resume disk access without coordinating with other application processes and this would permit it to make progress in the absence of conflicting access.

Our experience with developing and deploying sample applications on top of Minuet suggests that the availability benefits enabled by the use of such fine-grained recovery actions are certainly worth the extra implementation effort, which we believe to be relatively small. The chunkmap application was initially implemented on top of conventional locking using 327 lines of C code and extending the implementation to operate on top of Minuet required adding only 15 lines of code to handle the *EBADSESSION* notifications.

7 Related Work

Concurrency control has been extensively studied in the operating systems, distributed systems, and database communities. VMS [40] was among the first widely-available platforms to provide application developers

with the abstraction of a general-purpose distributed lock manager and today, DLMs are generally viewed as a useful building block for distributed applications.

Clustered and distributed filesystems [8, 10, 35–37] and relational databases [9] rely on locking or lease-based mechanisms to coordinate access to shared application state. Both sets of mechanisms make certain assumptions about timing, such as partially-synchronized clocks and bounded communication latency, in order to operate safely. These systems can directly leverage Minuet to ensure safe coordination of concurrent access to shared data on disk without assuming synchrony.

In web service data centers, distributed coordination services such as Chubby [20] and Zookeeper [15] 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 candidates. 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 majority voter set. Unlike our system, Chubby provides a hierarchical namespace and the ability to store small pieces of data, but these features are largely orthogonal to our approach. Chubby's *lock sequencer* mechanism allows servers to detect out-of-order requests submitted under an outdated lock and our timestamp-based *sessions* generalize this idea to support shared-exclusive locking. We also develop this notion further and observe that once we have the ability to reject inconsistent requests at the destination, very little is gained by enforcing strong consistency on replicated locking state and specifically, the use of an agreement protocol (e.g., Paxos [31]) may be more than necessary.

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 [38] 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 3.4 is a variation of timestamp-based concurrency control - a standard and well-known technique in relational 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 general SAN applications, where the data resides on application-agnostic block devices.

There have been several research projects tackling the intelligence/information gap between operating systems and storage systems [17, 25, 28]. These projects aim to achieve more expressive storage interfaces by exposing more information or adding more intelligence to storage devices. In our work, we identified and tackled safety problems in SANs by narrowing the intelligence gap between clustered applications and SAN storage devices.

Several earlier projects have investigated new approaches to concurrency control via functional extensions to storage devices. [29] proposes Dlocks as a new primitive for distributed mutual exclusion, whereby the lock acquisition state is maintained by the target devices themselves and manipulated by the initiators using a new SCSI command. Due to the inherent complexity of distributed locking, the lock management functionality has proven too difficult to implement in a SAN storage array and as a result, this mechanism did not attain wide acceptance among the storage device vendors. Follow-on work to the initial proposal presented a simplified scheme in form of DMEP [12]. In this scheme, storage devices expose an array of shared memory buffers holding the lock state and clients manipulate this state directly using simple atomic commands. The DMEP specification was implemented by a storage device vendor [2] and used by earlier versions of GFS [35].

Our work revisits the idea of device-assisted synchronization and is in line with these earlier efforts, but differs in several crucial respects. First, rather than extending the storage devices with lock management functions, we propose a more general synchronization primitive that supports a wider range of coordination techniques. In addition to "traditional" conservative locking, Minuet enables the use of optimistic concurrency control, which has been shown to reduce the synchronization overhead and deliver better performance for certain application workloads. As a result, Minuet enables a new degree of freedom in the design space of parallel SAN applications, enabling the developers to safely exploit the tradeoffs between synchronization overhead, access conflict rate, and application availability. Second, acquiring or releasing a lock in Minuet does not require explicit communication with the target storage device and instead, clients annotate outgoing I/O requests with the relevant synchronization state. This technique addresses the problem of delayed requests delivered under the protection of an outdated lock and thus enables SAN applications to guarantee safety despite arbitrary message delays, drifting clocks, and node failures. Finally, unlike prior proposals, our design does not require new SCSI commands and can be implemented within the confines of existing protocol standards.

Similar in spirit to this work, SCSI-3 Persistent Reserve [11] tries to address the safety problems in shared-disk environments by extending the storage protocol and target devices. Revoking a suspected node's reservation

typically necessitates a global decision on declaring the respective node faulty, which, in turn, requires majority agreement. Hence, SCSI-3 PR offers safety but not liveness in the presence of network partitions and massive node failures, while Minuet provides both.

8 Conclusion

This paper investigates a novel approach to concurrency control in SANs. Today, clustered SAN applications coordinate access to shared state on disks using strongly-consistent locking protocols, which are subject to safety and liveness issues in the presence of asynchrony and failures. To solve these problems, we augment target devices with a small amount of guard logic, which enables us to provide a property called session isolation and a relaxed model of locking which, in turn, provide a building block for distributed transactions. They also enable us to loosen the consistency requirements on distributed locking state, thus providing high availability despite failures and network partitions.

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

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments and our shepherd, James Bottomley, for his guidance.

References

[1] Brocade 5300 switch. `http://www.brocade.com`.

[2] Dot Hill and Sistina Software team up to improve computing performance for Linux users. `http://findarticles.com/p/articles/mi_m0EIN/is_2001_August_28/ai_776015%52`.

[3] HP remote insight lights-out edition II (QuickSpecs). `http://www.hp.com`.

[4] IETF RFC 3720: Internet small computer systems interface (iSCSI). `http://www.ietf.org`.

[5] iSCSI enterprise target v0.4.16. `http://iscsitarget.sourceforge.net/`.

[6] Open-iSCSI. `http://www.open-iscsi.org`.

[7] OpenDLM. `http://opendlm.sourceforge.net`.

[8] Oracle OCFS. `http://www.oracle.com`.

[9] Oracle real application clusters. `http://www.oracle.com`.

[10] Panasas PanFS. `http://www.panasas.com`.

[11] SCSI-3 block commands (draft proposed standard). `http://www.t10.org`.

[12] SCSI device memory export protocol (version 0.9.8). `http://www.t10.org`.

[13] SCSI Primary Commands - 4 (SPC-4), working draft. project T10/1731-D revision 17. `http://www.t10.org`.

[14] Survey finds SAN usage becoming mainstream. `http://findarticles.com/p/articles/mi_qa4137/is_200402/ai_n9362169/pg_1%`.

[15] ZooKeeper. `http://zookeeper.sourceforge.net`.

[16] Private communication with IBM Research, 2005.

[17] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS*, 1998.

[18] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD*, pages 23–34, 1995.

[19] T. Asaro. ESG analysis - the state of iSCSI-based IP SAN 2006. `http://www.netelligentgroup.com/articles/esgiscsi.pdf`.

[20] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[21] M. Carey, M. Franklin, and M. Zaharioudakis. *Fine-grained sharing in a page server OODBMS*. ACM, 1994.

[22] A. Ermolinskiy, D. Moon, B.-G. Chun, and S. Shenker. Deploying and evaluating an iSCSI-based implementation of Minuet. In *UC Berkeley EECS TR*, Jan 2009.

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

[24] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM TODS*, 22(3):315–363, 1997.

[25] G. R. Ganger. Blurring the line between OSes and storage devices. In *CMU SCS Technical Report CMU-CS-01-166*, 2001.

[26] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, 1989.

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

[28] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). In *SIGMOD Record*, Sept. 1998.

[29] C. J. S. Kenneth W. Preslan, Steven R. Soltis, M. O'Keefe, G. Houlder, and J. Coomes. Device locks: mutual exclusion for storage area networks. In *IEEE MSS*, pages 262–274, 1999.

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

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

[32] P. L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. DB Syst.*, 6(4):650–670, 1981.

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

[34] S. H. Phatak and B. R. Badrinath. Bounded locking for optimistic concurrency control. *Rutgers University TR DCS-TR-380*, 1999.

[35] K. W. Preslan, A. Barry, J. Brassow, M. Declerck, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe. Scalability and failure recovery in a linux cluster file system. In *USENIX ALS*, 2000.

[36] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. In *MSST 2003*, pages 207–218.

[37] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST 2002*, pages 231–244.

[38] J. Speer and M. Kirchberg. D-ARIES: A distributed version of the ARIES recovery algorithm. *ADBIS Research Communi.*, 2005.

[39] R. Treiber and J. Menon. Simulation study of cached RAID5 designs. In *HPCA*, page 186, Washington, DC, USA, 1995. IEEE.

[40] J. W. E. Snaman and D. W. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, Sept. 1987.

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI 2002*, pages 255–270.

[42] M. Zaharioudakis, M. Carey, and M. Franklin. Adaptive, Fine-Grained Sharing in a Client-Server OODBMS: A Callback-Based Approach. *ACM Trans. on Database Syst.*, 22(4):570–627, 1997.