Minuet: Rethinking Concurrency Control in Storage Area Networks

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 [1] 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 majority (80%) have deployed a storage-area network in their organizations, and 26% of the respondents report have deployed five or more SANs [2]. 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 to their customers [3–6].

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 [7], OCFS [8], PanFS [9], GPFS [10], Lustre [11], Xsan [12]) or a relational database (Oracle RAC [13]) 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) [7]. Traditional DLM 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 closer look at the semantics of traditional 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 of access to shared state on disk in the presence of process 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 is based on augmenting target storage 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 isolation*. The guard logic enables a novel *optimistic* approach to concurrency control in SANs and can be used to make existing protocols safe in the face of arbitrarily delayed message delivery, drifting clocks, crash process failures, and network partitions. In addition, session isolation provides a foundational primitive for implementing more complex coordination semantics, such as *serializable transactions*, and we demonstrate one such protocol.

We also 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 logic 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.

Unlike existing services for fault-tolerant distributed coordination such as Chubby [14] 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 replicas. To demonstrate the practical feasibility of our approach, we implemented two sample applications — distributed chunkmap and 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 explored extensively in the context of database management systems (DBMS) and are well-understood. In particular, techniques such as callback locking, optimistic 2-phase locking, and adaptive callback locking [16–20] have been proposed to enable safe coordination and efficient client-side 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 process liveness status. Hence, a conservative DLM service that enforces strict mutual exclusion has traditionally been viewed as the only practical method for disk access coordination for clustered 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 safely problems associated with traditional DLMs and enables a very different approach to 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 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 of Minuet and two sample clustered 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 (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 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 application nodes and target storage devices.

The goal is to provide fully-decentralized access to shared application state and in principle, any SAN-attached application 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 can be seen as 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 application cluster and in the event of a node failure, subsequent requests can be redirected to another processor with minimal interruption of service. In contrast, a node failure in the server-attached model may render some portions of the dataset temporarily or permanently unavailable.

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 [7]. A typical lock service, such as OpenDLM [21], exposes a generalized notion of a *resource*, an abstract application-level entity that requires access coordination, and attempts to provide the guarantee of *mutual exclusion* [22] - no two processes may simultaneously hold conflicting locks on the same resource.

2.2 Safety and liveness problems in SANs

In principle, DLM-based mutual exclusion offers sufficient mechanism to guarantee safe access to shared application state on disk. 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 *process failures* and *asynchrony*. The following examples illustrate the nature of the problem.

Scenario 1: 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 pro-

tection of an exclusive lock, while C_2 wants to read the contents of S into a local memory buffer and is waiting for a shared lock on S. C_1 submits $WRITE(offset = 3, length = 5, data)$ to D, but crashes before hearing a response and the lock manager correctly detects the failure (e.g., via a heartbeat mechanism) and reacts by reclaiming the exclusive lock and granting it in shared mode to C_2. That client proceeds to reading S from disk and submits $READ(offset = 0, length = 5)$ to D, which returns old data. Next, C_1's delayed $WRITE$ request reaches the disk and overwrites the data at offsets $[3-7]$, after which C_2 issues $READ(offset = 5, length = 5)$. Note that although each individual I/O request is processed by D as an atomic unit, this scenario 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 [23] 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 has no way of coordinating 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: 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 [24] 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 logs (also on shared disks) that record Redo and/or Undo information for all updates 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 flushed to the snapshot prior to failure. An essential underlying assumption 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 reach the snapshot or the log and data corruption may occur if this assumption is violated.

Ensuring data safety in a shared-disk environment has traditionally required introducing a set of *partial synchrony assumptions*, such as bounded network propagation delays and clock drift rates, that enable the use of reliable heartbeat-driven failure detectors and leases. Fundamentally, 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 and account for worst-case switch queuing delays and request buffering at the host. 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 [25].

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 mechanisms such as state machine replication [26] and Paxos [27]. Alternatively, a single lock manager instance may be elected dynamically [28–30] 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) [31, 32], SAN fabric fencing [33], and SCSI-3 PR [34] 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 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, 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 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 ap-

plications. Traditionally, DLMs tend to treat shared application resources as purely abstract entities enforce 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 [35] 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 using the term *session isolation*.

After defining this correctness property in formal terms, we describe the protocol machinery for enforcing session isolation on a single shared resource and then demonstrate how more complex and useful application semantics, such as distributed transactions, can be supported using session isolation as a foundational building block. Lastly, we address the issue of fault tolerance and present a mechanism for loosely-consistent replication of locking state.

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 (1) Issuing READ/WRITE commands to $R.owner$; (2) Acquiring and releasing locks on $R.resID$.

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** relinquishes the shared lock (i.e., downgrades to **NoLock**). Analogously, by acquiring an **Excl** lock, 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 a client's local execution history, we define **Sessions(t, C, R)** to be the set of **C**'s active sessions to **R** at time **t**, which is determined solely by the sequence of 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.*

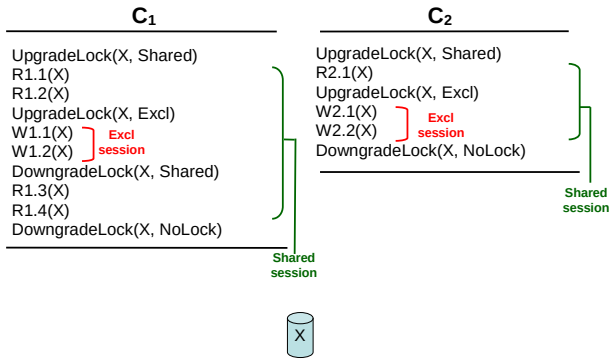

Figure 1: Concurrent request streams to a shared resource X from two client processes, C_1 and C_2. $R_{i.j}$ denotes the j-th $READ$ operation from client i and $W_{i.j}$ represents a $WRITE$ operation, accordingly.

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

Definition 2. *If a client process **C** issues at time **t** a disk request **r** that operates on **R**, we say that **r belongs to** a 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:*

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

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

for some 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 a 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 issues 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 satisfy session isolation:

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

An execution history that causes X 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.

Our core approach is inspired by earlier work on bridging the intelligence gap between applications and block storage devices. [36, 37]. We augment SAN-attached disks with a small amount of application-independent logic, which we call a *guard*, that enforces the session isolation invariant on the stream of incoming I/O requests. We associate a *session identifier* (*SID*) with every lock granted to a client and modify the storage protocol stack on application nodes to annotate all outgoing disk requests with the current *SID* for the respective resource. Below, we refer to this annotation as a *request capsule*.

The guard logic at target storage devices evaluates incoming requests based on the attached *SID* and, for each request, determines whether its acceptance would violate session isolation. All such requests are dropped from the input queue and the originating client process is notified via a special error code *EREJECTED*. From an application developer's point of view, session rejection appears as a failed I/O request along with an exception notification from the lock service indicating that a lock on the respective resource is no longer valid.

The guard logic situated at the storage 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 Minuet, the primary purpose of the lock service is ensuring an efficient assignment of session identifiers to clients that minimizes the aggregate rate of session 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 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 endpoints and allow application developers to trade off lock service availability, synchronization overhead, and I/O performance under heterogeneous data access patterns.

3.2 Enforcing session isolation

Minuet uses a simple timestamp-based mechanism to enforce session isolation 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, the client *proposes* a session timestamp to the Minuet lock manager. These proposals 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: $\langle T.incNum.cliID\rangle$, where *cliID* uniquely identifies the client and *incNum* is the client's *incarnation number* - a monotonic counter ensuring uniqueness across crashes.

The basic locking protocol proceeds as follows: every 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, C 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 $MaxT_s(C,R)$.

The client then sends an $UpgradeLock$ request to the Minuet 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 $MaxT_s(C,R)$ 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. The receipt of this message marks the start of a shared session and the client initializes its session identifier (denoted $R.cliSID$) as follows: $R.cliSID := \langle ProposedT_s, ProposedT_x\rangle$. It also sets the current session type (denoted $R.cliSType$) to *Shared*.

Acquisition of an *Exclusive* lock (which includes up-

grading from *Shared* to *Exclusive*) proceeds analogously except that clients increment the T_x value in the proposal 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 client sets $R.cliSType := Exclusive$.

When a client issues a disk request operating on a resource *R*, it augments the request message with a *request capsule* that identifies the affected resource and carries a tuple of the form $\langle R.resID, R.cliSID, R.cliSType \rangle$.

For each resource *R*, its owner maintains a small amount of metadata, which we call the *owner session identifier* (*R.ownSID*), initially set to $\langle T_s = 0, T_x = 0 \rangle$. Upon receipt of an I/O request from a client, the owner invokes the guard logic, which evaluates the request capsule against *R.ownSID* to determine whether session isolation would be preserved by accepting the request. Capsule evaluation proceeds as follows: If *R.cliSType* specifies a *Shared* session, the owner **accepts** and enqueues the request iff $R.cliSID.T_x \geq R.ownSID.T_x$ and **rejects** it otherwise. Likewise, if *R.cliSType* specifies an *Exclusive* session, the owner accepts the request iff $R.cliSID.T_s \geq R.ownSID.T_s$ and $R.cliSID.T_x \geq R.ownSID.T_x$.

Upon acceptance, the owner updates its session identifier, setting $R.ownSID.T_s$ to be the maximum of $R.ownSID.T_s$ and $R.cliSID.T_s$ and setting $R.ownSID.T_x$ to the maximum of $R.ownSID.T_x$ and $R.cliSID.T_x$. Otherwise, the request is discarded and an $EREJECTED$ response is sent to the client, together with a *response capsule* that carries $\langle R.ownSID \rangle$.

Upon receipt of $EREJECTED$, the Minuet client examines the response capsule and notifies the application process that its lock on *R* is no longer valid. An *Exclusive*-mode lock is downgraded to *Shared* if $R.ownSID.T_s > R.cliSID.T_s$ (since that indicates interruption of an exclusive session) and a *Shared* lock is further downgraded to *NoLock* if $R.ownSID.T_x > R.cliSID.T_x$ (since in this case, a conflicting exclusive-session request has been accepted). The client also updates $MaxT_s(C,R)$ and $MaxT_x(C,R)$ to reflect the most recent timestamp values seen by the owner.

A diagram illustrating the basic locking protocol and a formal correctness argument demonstrating that the protocol and the guard logic described above ensure session isolation can be found in [38]. 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 got accepted with a session identifier $\langle R.cliSID.T_s^1, R.cliSID.T_x^1 \rangle$. Observe that due to global uniqueness of session proposals, the owner would subsequently accept an exclusive-session request from C_2 with a session identifier $\langle R.cliSID.T_s^2, R.cliSID.T_x^2 \rangle$ *only if* $R.cliSID.T_x^2$ is strictly greater than $R.cliSID.T_x^1$. In this case, subsequent shared-session requests from C_1 would get rejected and session isolation would be preserved. A similar argument demonstrates that no two exclusive-session requests can be interleaved by a conflicting request from another client.

3.3 Supporting transactional semantics

3.3.1 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 commit a transaction, a client appends to the log a sequence of Redo records that concisely describe its updates, after which a special *Commit* record is force-appended. Prior to releasing a lock on a dirty resource *R*, its holder flushes all committed updates to the snapshot of *R*, which ensures that the next reader observes the effects of every committed transaction. If a client crashes during a transaction, the recovery process examines its portion of the log and restores the affected resource snapshots to a consistent state by replaying or rolling back updates from the log.

To support transactions, Minuet relies on this well-understood and widely-used mechanism, while extending it with the use of the guard logic 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 [24]. 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 logic at target devices to provide session isolation and protect the state on disk from the effects of arbitrarily-delayed WRITEs 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 logic 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 [39].

(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.

3.3.2 Basic transaction protocol

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

To support transactions, we extend the basic session isolation machinery described in Section 3.2 with an additional piece of state called a *commit session identifier (CSID)*, which has the following format: $\langle cliID, xactID \rangle$. We also extend the request capsule to carry two commit session identifiers with each disk request, denoted *compareCSID* and *setCSID*, and both are set to NIL unless specified otherwise. For each shared resource R, the owner device maintains a commit session identifier (*R.ownCSID*) in addition to *R.ownSID*. Upon receiving a disk request, the guard logic examines the capsule and rejects the request if $R.compareCSID.cliID \neq R.ownCSID.cliID$ or if $R.compareCSID.xactID < R.ownCSID.xactID$. A capsule is accepted only if its *compareCSID* and *cliSID* both pass verification and upon completing the request, the owner device sets $R.ownCSID := R.setCSID$. If verification fails, the owner responds with $EREJECTED$ and a response capsule carrying the tuple $\langle R.ownSID, R.ownCSID \rangle$.

In Minuet, transactions proceed in five stages: *Begin*, *Read*, *Update*, *Verify*, and *Commit* and we illustrate them using high-level pseudocode in [38]. During one-time initialization, Minuet's transaction service at client C locks the client's log in *Exclusive* mode. To begin a new transaction T, the client selects a new transaction identifier ($curXactID$) via a monotonically increasing 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 read by the transaction (denoted $T.ReadSet$) and reads the corresponding data from remote disks into local memory buffers. In the *Update phase* that follows, the process applies the desired set of updates locally and communicates a description of these 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 *Verification phase* serves a dual purpose: to verify the validity of client's sessions (and hence, the accuracy of cached data) and to prepare the elements of $T.WriteSet$ for committing. For each resource in $T.ReadSet \cup T.WriteSet$, client sends a special $VERIFY$ disk request[1] to its owner, whose sole purpose is to transport a capsule and invoke the guard logic at the device. $VERIFY$ requests for elements of $T.WriteSet$ carry $\langle compareCSID = NIL, setCSID = \langle C, curXactID \rangle \rangle$ in the request capsule. If all resource sessions pass verification, the transaction enters the final *Commit phase*, during which a *CommitXact* record is force-appended to the log.

The protocol outlined above ensures transaction isolation, identifying cases of conflicting access during the verification phase. Recall, however, that under the session isolation semantics, any I/O command, including operations on the log, may fail with $EREJECTED$ due to conflicting access from another client. This gives rise to several exception cases at various stages of transaction execution. For example, a client may receive an error while forcing a *CommitXact* record 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 [38].

3.3.3 Syncing updates to disk

After committing a transaction, a client C can flush its locally-buffered updates to R simply by issuing a sequence of corresponding $WRITE$s to *R.owner*. Each such command carries the following parameters in the attached capsule: $\langle R.compareCSID = \langle C, syncXactID \rangle, R.setCSID = \langle C, syncXactID \rangle \rangle$, 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 $\langle R.compareCSID = \langle C, syncXactID \rangle, R.setCSID = NIL \rangle$ in the capsule. This request causes the storage device to reset *R.ownCSID* to NIL, effectively marking the disk image of R as "clean". Lastly, C appends to its log an $UpdateSynced$ record of the form $\langle R, syncXactID \rangle$.

[1] Minuet implements $VERIFY$ requests as zero-length $WRITE$s.

3.3.4 Lazy transaction recovery

A client C can initiate transaction recovery when its disk request on some resource R fails with $EREJECTED$ and a non-NIL $R.ownCSID$ value $\langle C_F, xactID \rangle$ is returned in the response capsule. 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 process that tries to repair the disk image. 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 repair disk requests sent to the owner during this phase specify $\langle R.compareCSID = R.ownCSID, R.setCSID = R.ownCSID \rangle$ in the request capsule. Finally, after reapplying all missing updates, C completes recovery by sending a zero-length $WRITE$ request to the owner with $\langle R.compareCSID = R.ownCSID, R.setCSID = NIL \rangle$ in the request capsule. A more detailed discussion of transaction recovery in Minuet can be found in [38].

3.4 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 of the manager instances is reachable [2].

To support manager replication, we extend the basic locking protocol presented in Section 3.2 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[3].

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 clustered applications.

4.1 Core Minuet modules

Client-side library (5440 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 the appropriate request capsules 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 (4285 LoC): The lock manager process grants and revokes locks using the timestamp mechanism of Section 3.2 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 capsule propagation within the framework of iSCSI [40], a widely-used protocol 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 (*linux/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* command, which enables the Minuet client library to specify request capsules for outgoing disk requests and to retrieve response capsules.

The bottom-level driver implements TCP encapsulation of SCSI commands and our current prototype builds upon the Open-iSCSI Initiator driver [41] v2.0-869.2. We used the Additional Header Segment AHS) feature of iSCSI to attach Minuet capsules to SCSI command PDUs and defined a new AHS type for this purpose.

Our storage backend is based on the iSCSI Enterprise

[2] 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.

[3] As a performance optimization, we allow $UpgradeLock$ requests to specify an *implicit downgrade* for an earlier timestamp.

Target driver [42] v0.4.16, which exposes a local block device to remote clients via iSCSI. We extended it with the guard logic, which examines incoming command PDUs and makes an accept/reject decision based on the capsule content. 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 functionality 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 14341 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 exclusive lock on the respective block from Minuet prior to reading it from disk and releases the lock after writing the modified version.

Distributed B-Tree (3345 LoC): To demonstrate the feasibility of serializable transactions, we implemented a distributed *B-link tree* [43] (a variant of *B+-tree*) on top of Minuet. Our implementation provides *Insert*, *Delete*, *Update*, and *Search* operations based on the transaction protocol presented in Section 3.3.2. 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 exclusive mode and logs the updates (*Update phase*). Lastly, the client verifies the sessions on all blocks in the chain (*Verification phase*) and commits the transaction only if all sessions are valid. If a transaction aborts due to loss of session to a B-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 locks (and the content of cache buffers) across transactions and stale cache entires are detected and invalidated during the verification phase.

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 of the sample applications atop Minuet and provide comparison with traditional strongly-consistent locking.

	Storage servers	Lock managers	Clients
# Nodes	4	5	32
CPU	3GHz Xeon	850Mhz Pentium III	
RAM	2GB	512MB	
DISK	10K RPM SCSI	7200 RPM IDE	

Table 1: Hardware specification of the cluster

5.1 Experimental setup

For our experiments, we emulated a SAN environment with 41 Emulab [44] nodes, interconnected via 100Mbps links. Detailed hardware specification is given in Table 1.

We allocated four storage nodes that provided 2GB of logical disk space, equally striped across the nodes. The remaining machines were assigned to client processes and Minuet lock manager processes. Client instances ran across 32 nodes and they saturated neither CPU nor RAM.

In our experiments, we measured application operation goodput, the number of successful application-level operations per second, varying the number of clients (i.e. offered load) under the following two locking scenarios[4]:

***strong(x)*:** A strongly-consistent locking protocol. Each client must get permissions from a majority (x) of lock manager processes. $2x-1$ nodes were dedicated to run $2x-1$ lock manager processes.

***weak-own*:** An extreme form of weakly-consistent locking. Each client has its own lock manager and does not coordinate with other clients.

We also considered two forms of workload:

***uniform*:** In the chunkmap application, each operation selects the block to modify uniformly at random. In the B+-tree application, each operation chooses a key to access uniformly at random.

***skewed(x/y)*:** This is a hotspot workload. y% operations touch x% of the entire blocks or entire key space.

5.2 Distributed chunkmap

We configured the block size to 4 KB and ran experiments in which each client modified blocks for five minutes.

Figure 2 shows the aggregate operation goodput under the *uniform* workload. Since there are a large number of blocks (500K blocks) in the storage node, this result represents a low-contention scenario. We observe that the weakly-consistent locking scheme shows slightly better performance, up to 32 clients, than the strong locking scheme with five lock managers. This result suggests that our approach has a potential in improving application goodput in scenarios while guaranteeing safety where the overall load is high, but contention for a single resource is

[4]Note that in our experiments, applications rely on Minuet to provide both modes of locking (i.e., $strong(x)$ and $weak-own$) and do not make use of any other synchronization facilities.

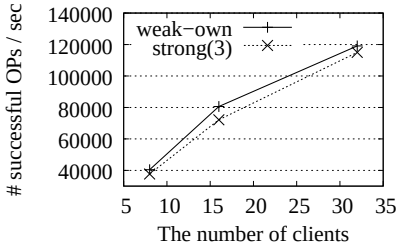

Figure 2: Goodput of the distributed chunkmap under the *uniform* workload

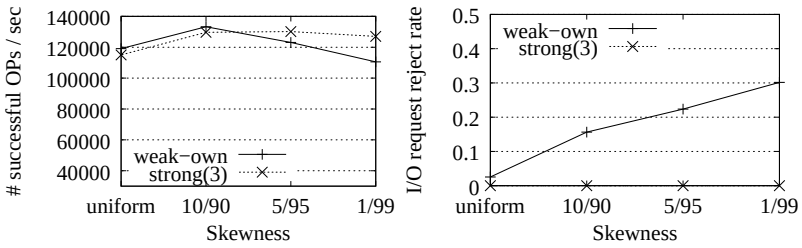

Figure 3: Goodput (left) and the rate of rejected I/O requests (right) observed by 32 clients varying the skewness of the workload. The fact that the reject I/O request rate is 0.1 means 10% of I/O requests are rejected.

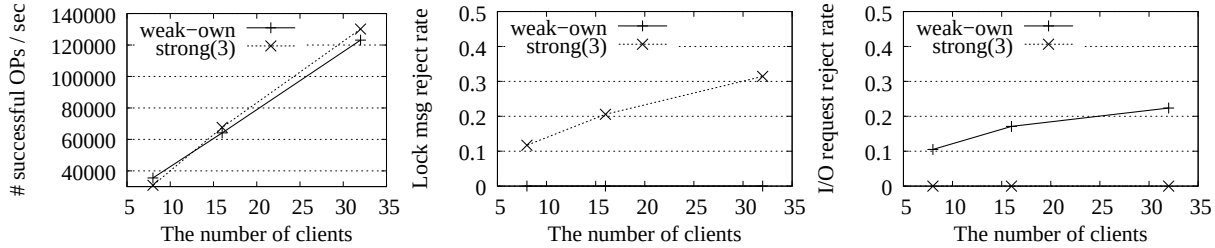

Figure 4: Goodput of the distributed chunkmap (left), the rate of denied lock requests (center), and the rate of rejected I/O requests (right) under the *skewed(5/95)* workload.

relatively rare. Moreover, even skewed workload scenario shows the weakly-consistent locking provides comparable performance to the strong locking scheme. (Figure 4 (left)) This is because clients in the *weak-own* scenario do not incur locking overhead (Figure 4 (center)), despite rejected I/O requests (mostly *READ* requests) at the storage nodes (Figure 4 (right)).

The rate of I/O rejection increases when a system has resource hotspots (Figure 3). However, in our experiments, weakly-consistent locking can still provide reasonable performance in such scenarios, since traditional strong locking also face increasing lock synchronization overhead.

We also ran experiments of a partitioined 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 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, 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.

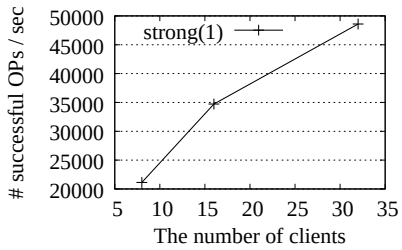

Figure 5: Goodput of the distributed B+-tree under the *uniform* workload

5.3 Distributed B-tree

We configured each tree node to 8KB to hold 150 keys and satellite values (e.g., a pointer to other nodes) in a single block. We pre-populated B-tree in our experiments; we inserted 200K keys a priori so that the tree height is three. After creating the distributed tree, we randomly issued insertion operations and search operations at the 80% probability and 20% probability, respectively. Target keys were chosen randomly.

Figure 5 shows the goodput of the distributed B-tree application with *strong(1)*. Both locking schemes are safe to use thanks to Minuet. With *strong(1)*, the application's goodput scales as the client workload increases. However, with *weak-own*, the goodput does not scale. From our debugging, it turns out that the problem happened because

only one client keeps sending transactions successfully since clients are greedy (i.e., send requests as fast as they can). This problem is a general problem in any optimistic concurrency control scheme. Due to time constraints, we were unable to perform a full-scale performance analysis and resolve this problem but we plan to address these issues in the future by exploring some arbitration between clients or by locally introducing some randomised delays between transactions.

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.

Practical feasibility: Our approach rests on the basic idea of extending network-attached storage arrays 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 or firmware and modifying existing storage protocols to carry some additional state, which may raise concerns about the feasibility of our approach.

We acknowledge that Minuet assumes functionality that does not presently exist in standard disk hardware and, consequently, faces a non-trivial barrier to deployment. However, we observe that the proposed changes are very incremental in their nature and can be easily implemented within the confines of a traditional SAN access protocol such as iSCSI. 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 concurrency protocols, enabling a choice between optimism and strict coordination.

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 considered common [45]. Assuming 128KB application block size, our design incurs a storage overhead of 0.01%.

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 [46, 47].

Different programming model: Another concern is that Minuet introduces an alternative 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 inconsistent 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 $EREJECTED$ due to inconsistent lock state.

We observe that while I/O 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 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 $EREJECTED$ and *ForcedDowngrade* notifications.

7 Related Work

Concurrency control has been extensively studied in the operating systems, distributed systems, and database communities. VMS [48] was among the first widely-available operating systems to provide application developers with the abstraction of a general-purpose DLM. Since then, DLMs have been widely adopted for various purposes and today, they are viewed as a useful general-purpose build-

ing block for distributed applications and middleware.

Clustered filesystems (GFS [7], OCFS [8], PanFS [9], GPFS [10], Lustre [11], Xsan [12]) and relational databases (Oracle RAC [13]) rely on a distributed lock manager to coordinate parallel access to application data, metadata, and logs residing on shared disks. OpenDLM [21] is a widely-adopted general-purpose DLM implementation for Linux, currently used by GFS [7] and other clustered filesystems.

In web service data centers, distributed locking services such as Chubby [14] 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 Bigtable [49] 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 majority voter set. 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 approach. Chubby's *lock sequencer* mechanism allows servers to detect out-of-order requests submitted under the protection of 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 lock management state and specifically, the use of an agreement protocol (e.g., Paxos [27]) may be more than necessary.

Concurrency control and transaction mechanisms have been extensively studied in databases. In addition to database locking protocols mentioned in Section 1, we discuss other relevant database systems. ARIES [50] 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 [24] 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 [51] 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.

There have been several research projects tackling intelligence/information gap between operating systems and storage systems [36, 37, 52–55]. The projects aim to achieve more expressive storage interfaces by exposing more information or adding more intelligence to storage devices. Object-based storage introduces objects as storage resources [52]. Active disks execute downloaded generic code [37, 53]. ExRAID exposes performance and failure information and I-LFS extends a log structured file system by utilizing the information for better performance, flexibility, and reliability [55]. Track-aligned extents explores the benefits of exposing disk characteristics [54]. Our approach is in line with these research projects. In our work, we identified and tackled safety problems in SANs by narrowing the intelligence gap between clustered applications and SAN storage devices.

Similar in spirit to this work, SCSI-3 Persistent Reserve [34] tries to address the safety problems caused by inconsistent requests by extending the storage protocol and target devices. Typically, revoking a suspected node's reservation necessitates a global decision on declaring the respective process 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 our approach 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, but they are subject to safety and liveness problems in the presence of asynchrony and failures; strict mutual exclusion guarantees are neither sufficient nor necessary for application-level correctness.

To solve safety problems, we augment SAN target devices with a small amount of logic called a guard, which enables us to provide a property called session isolation and a relaxed model of locking. These, in turn, provide a foundational building block for more complex and useful application semantics such as transactions. We also show that this block enables us to loosen the consistency semantics of a distributed lock service, 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 techniques and protocols we presented. Our evaluation suggests that distributed applications built atop Minuet enjoy good performance and availability, while guaranteeing safety.

References

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

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

[3] EMC ControlCenter SAN manager. http://www.emc.com/products/detail/software/san-manager.htm.

[4] Storage area network products from HP StorageWorks. http://h18006.www1.hp.com/storage/networking/index.html.

[5] IBM storage area network: Overview. http://www-03.ibm.com/systems/storage/san/.

[6] NetApp fibre channel SAN solutions. http://www.netapp.com/us/products/protocols/fc-san/.

[7] Matthew O'Keefe and Paul Kennedy. Enterprise data sharing with Red Hat Global File System. http://www.redhat.com/magazine/009jul05/features/gfs_overview/, 2005.

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

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

[10] F. Schmuck and R. 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.

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

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

[13] Oracle real application clusters. http://www.oracle.com/technology/products/database/clustering/index.htm%l.

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

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

[16] M.J. Carey, M.J. Franklin, and M. Zaharioudakis. *Fine-grained sharing in a page server OODBMS*. ACM New York, NY, USA, 1994.

[17] M.J. Carey, M.J. Franklin, M. Livny, and E.J. Shekita. Data caching tradeoffs in client-server DBMS architectures. *ACM SIGMOD Record*, 20(2):357–366, 1991.

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

[19] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 23–34, 1995.

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

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

[22] Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC*, pages 51–60, 1998.

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

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

[25] Private communication with IBM Research, 2005.

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

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

[28] 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.

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

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

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

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

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

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

[35] 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.

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

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

[38] Anonymized. Minuet: Rethinking concurrency control in storage area networks (technical report). In *Available upon request from FAST'09 Program Co-Chairs*, 2008.

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

[40] IETF RFC 3720: Internet small computer systems interface (iSCSI). http://www.ietf.org/rfc/rfc3720.txt/.

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

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

[43] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[44] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad

Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[45] OCFS best practices. `http://oss.oracle.com/projects/ocfs/dist/documentation/UL_best_practice%s.txt`.

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

[47] 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.

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

[49] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of OSDI*, 2006.

[50] 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.

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

[52] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. of ASPLOS*, 1998.

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

[54] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, January 2002.

[55] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proc. of USENIX 20002 Annual Tech. Conf.*, 2002.