Minuet – Rethinking Concurrency Control in Storage Area Networks

FAST '09

Andrey Ermolinskiy (U. C. Berkeley)
Daekyeong Moon (U. C. Berkeley)
Byung-Gon Chun (Intel Research, Berkeley)
Scott Shenker (U. C. Berkeley and ICSI)

Storage Area Networks – an Overview

- Storage Area Networks (SANs) are gaining widespread adoption in data centers.

- An attractive architecture for clustered services and data-intensive clustered applications that require a scalable and highly-available storage backend. Examples:
  - Online transaction processing
  - Data mining and business intelligence
  - Digital media production and streaming media delivery

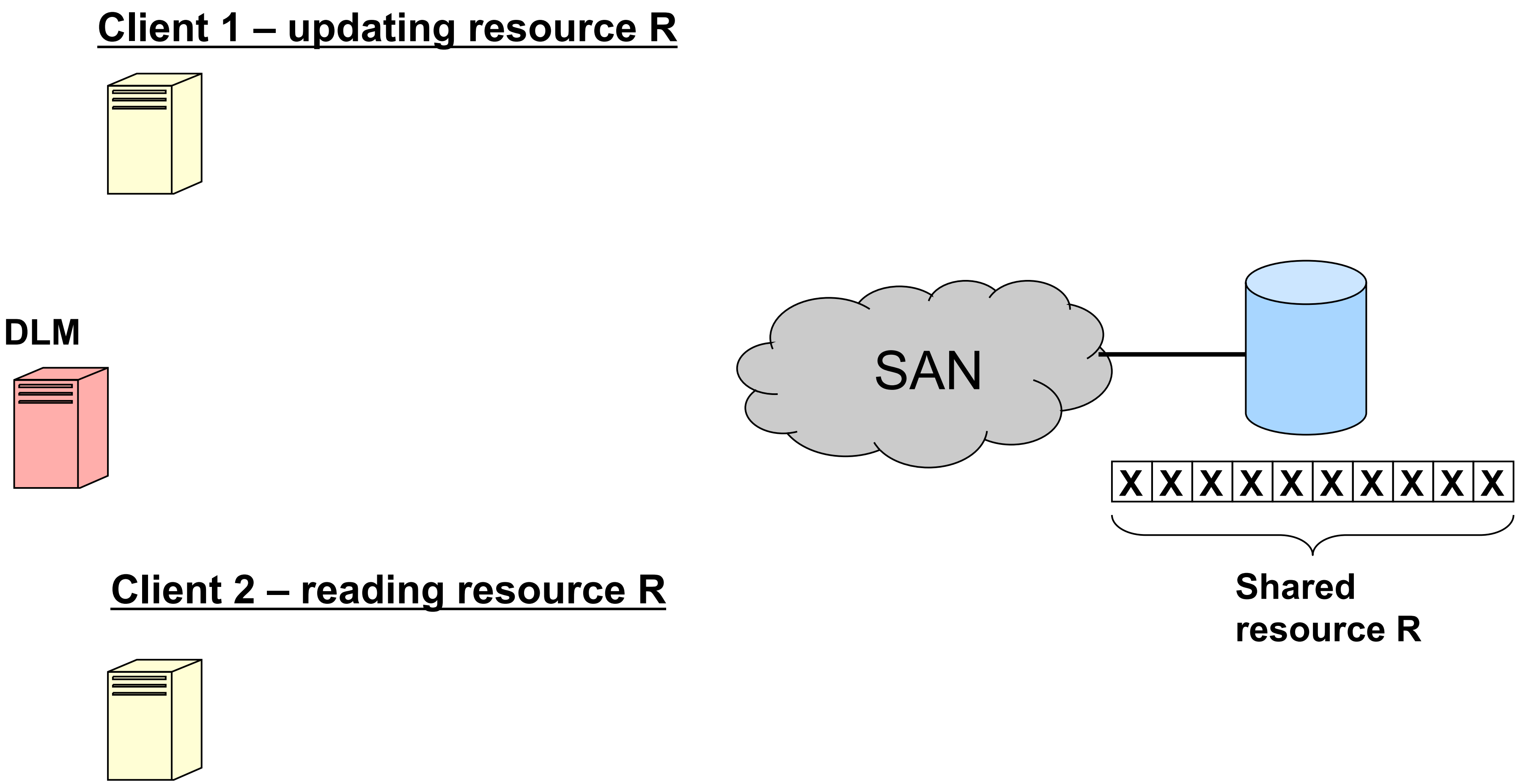
Clustered SAN applications and services

- One of the main design challenges: ensuring safe and efficient coordination of concurrent access to shared state on disk.
- Need mechanisms for distributed concurrency control.
- Traditional techniques for shared-disk applications: distributed locking, leases.

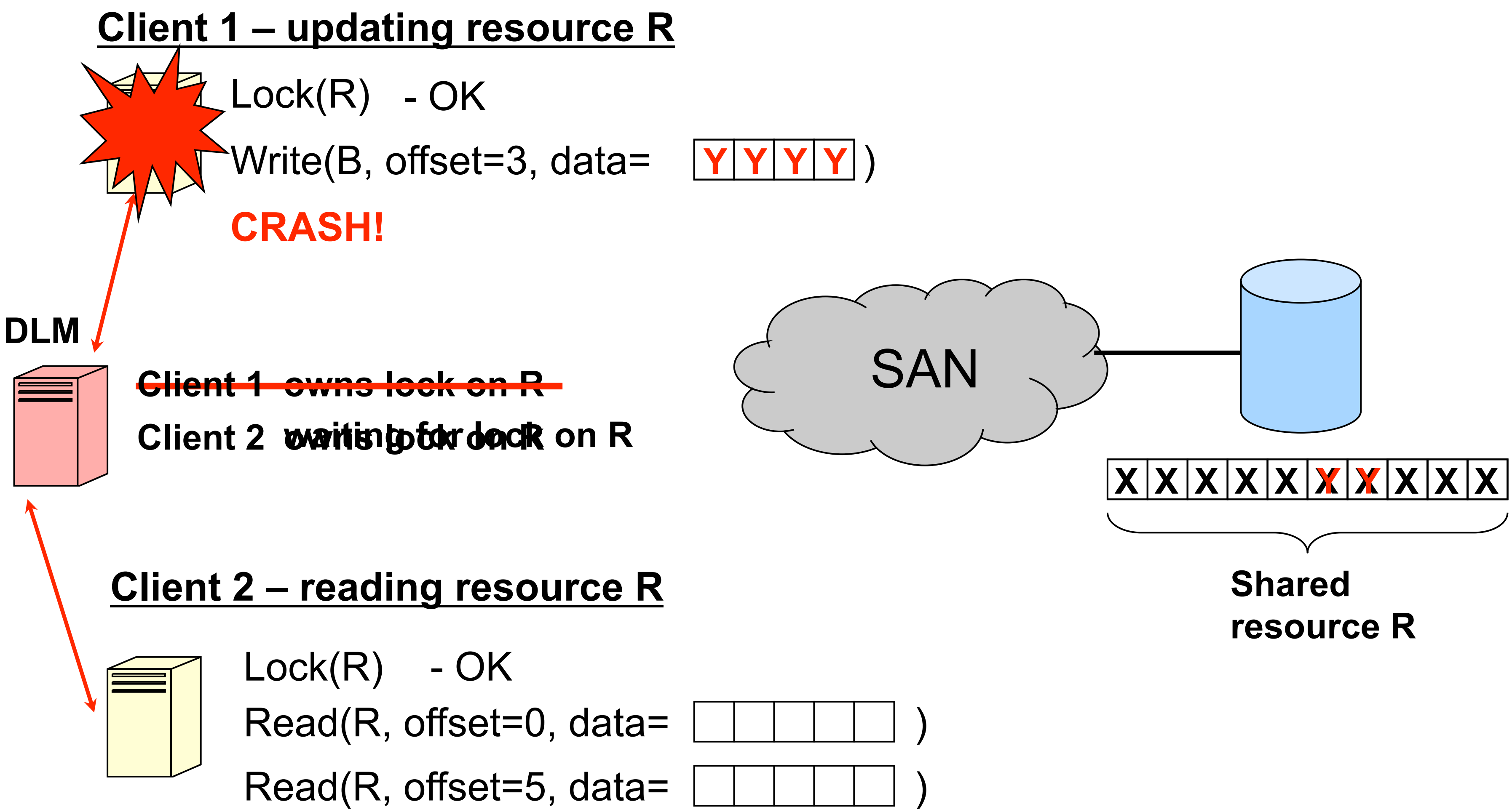
Limitations of distributed locking

- Distributed locking semantics do not suffice to guarantee correct serialization of disk requests and hence do not ensure application-level data safety.

Data integrity violation: an example

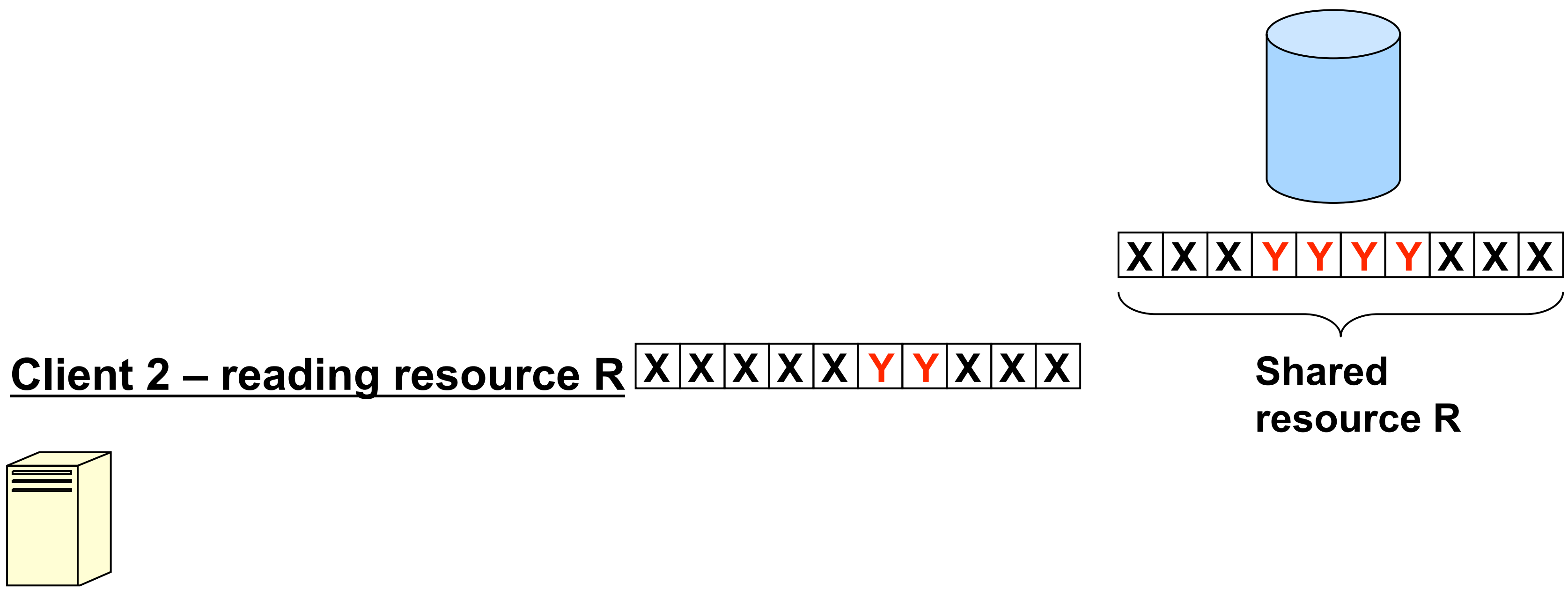


Data integrity violation: an example



Data integrity violation: an example

- Both clients obey the locking protocol, but Client 1 observes only partial effects of Client 2's update.
- Update atomicity is violated.



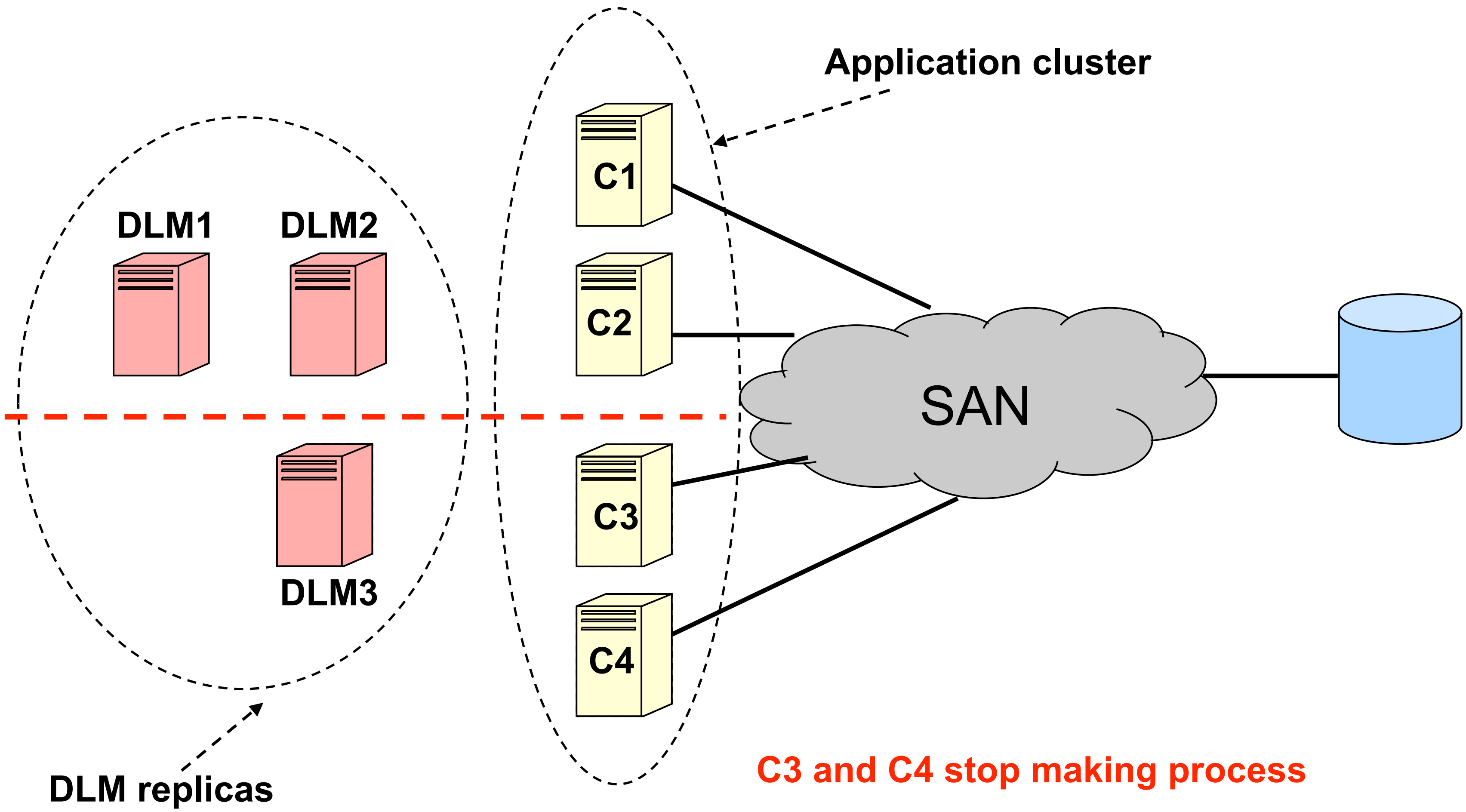
Availability limitations of distributed locking

- The lock service represents an additional point of failure.
- DLM failure → loss of lock management state → application downtime.

Availability limitations of distributed locking

- Standard fault tolerance techniques can be applied to mitigate the effects of DLM failures
  - State machine replication
  - Dynamic election
- These techniques necessitate some form of global agreement.
- Agreement requires an active majority
  - Makes it difficult to tolerate network-level failures and large-scale node failures.

Example: a partitioned network



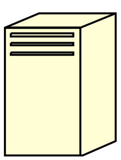
Minuet overview

- Minuet is a new synchronization primitive for shared-disk applications and middleware that seeks to address these limitations.
    - `Guarantees safe access to shared state in the face of arbitrary asynchrony
        - Unbounded network transfer delays
        - Unbounded clock drift rates
    - Improves application availability
        - Resilience to network partitions and large-scale node failures.

Our approach

- A “traditional” cluster lock service provides the guarantees of mutual exclusion and focuses on preventing conflicting lock assignments.
- We focus on ensuring safe ordering of disk requests at target storage devices.

<u>Client 2 – reading resource R</u>



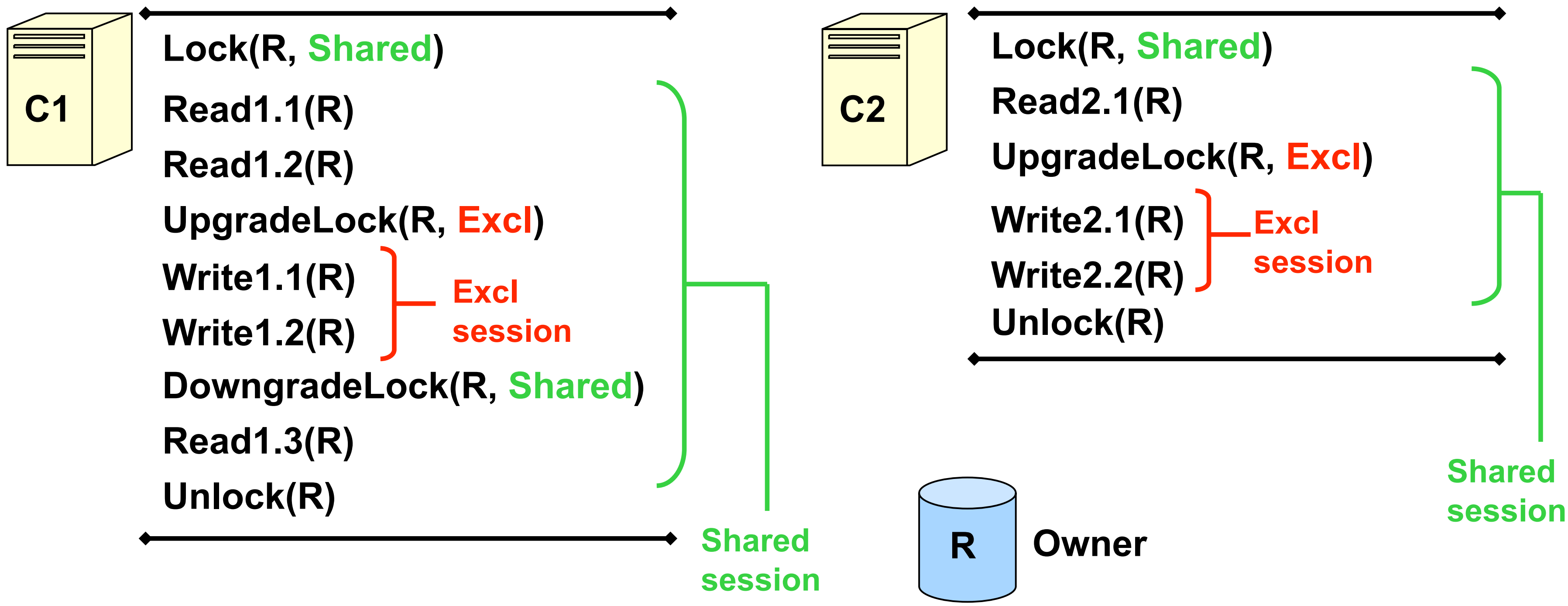
Lock(R)

Read(R, offset=0, data= □□□□□)

Read(R, offset=5, data= □□□□□)

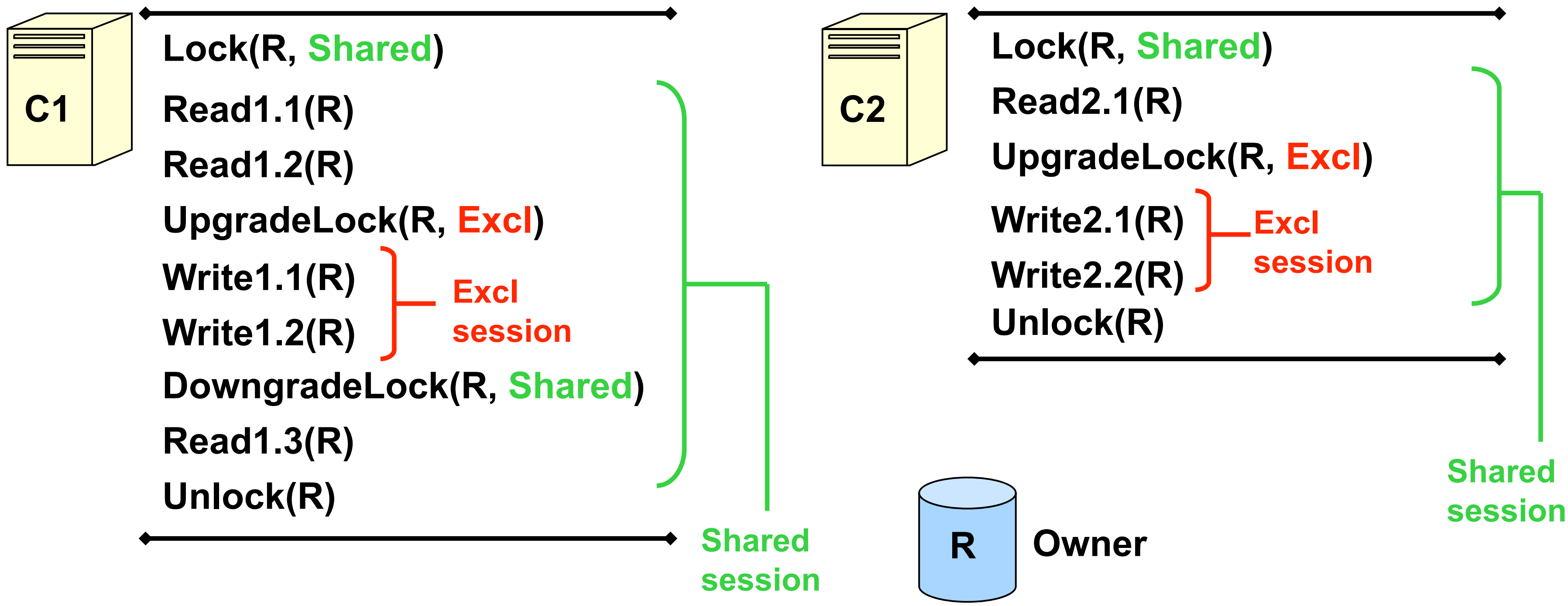
Unlock(R)

Session isolation



- *Session isolation*: R.owner must observe the prefixes of all sessions to R in strictly serial order, such that
  - No two requests in a shared session are interleaved by an exclusive-session request from another client.

Session isolation



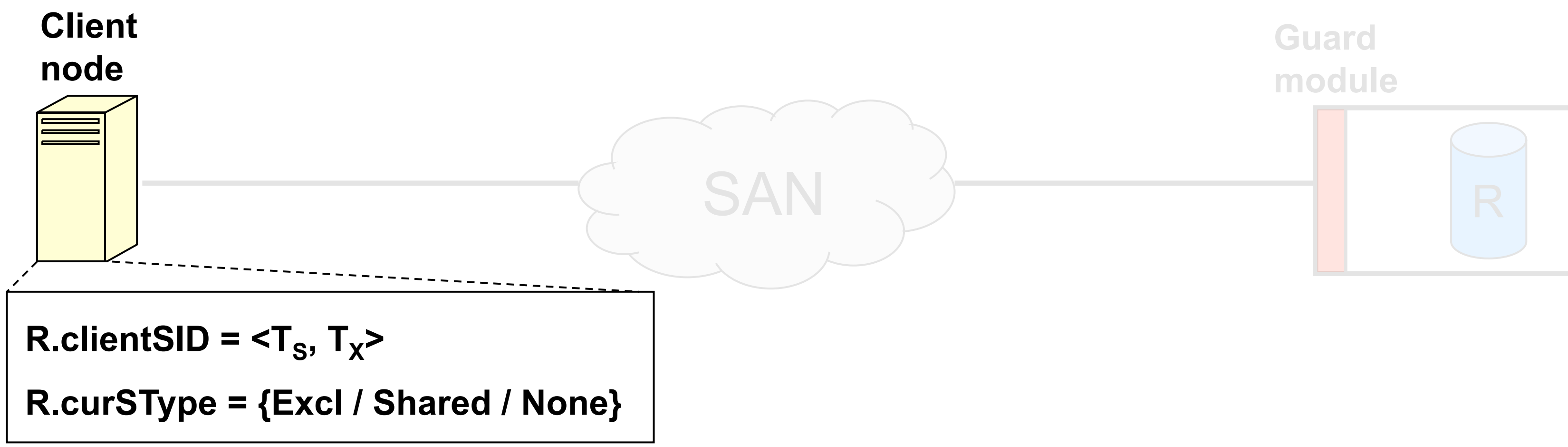
- *Session isolation*: R.owner must observe the prefixes of all sessions to R in strictly serial order, such that
  - No two requests in an exclusive session are interleaved by a shared- or exclusive-session request from another client.

Enforcing session isolation

- Each session to a shared resource is assigned a globally-unique session identifier (SID) at the time of lock acquisition.
- Client annotates its outbound disk commands with its current SID for the respective resource.
- SAN-attached storage devices are extended with a small application-independent logical component ("*guard*"), which:
  - Examines the client-supplied session annotations
  - Rejects commands that violate session isolation.

Enforcing session isolation

Enforcing session isolation



Enforcing session isolation

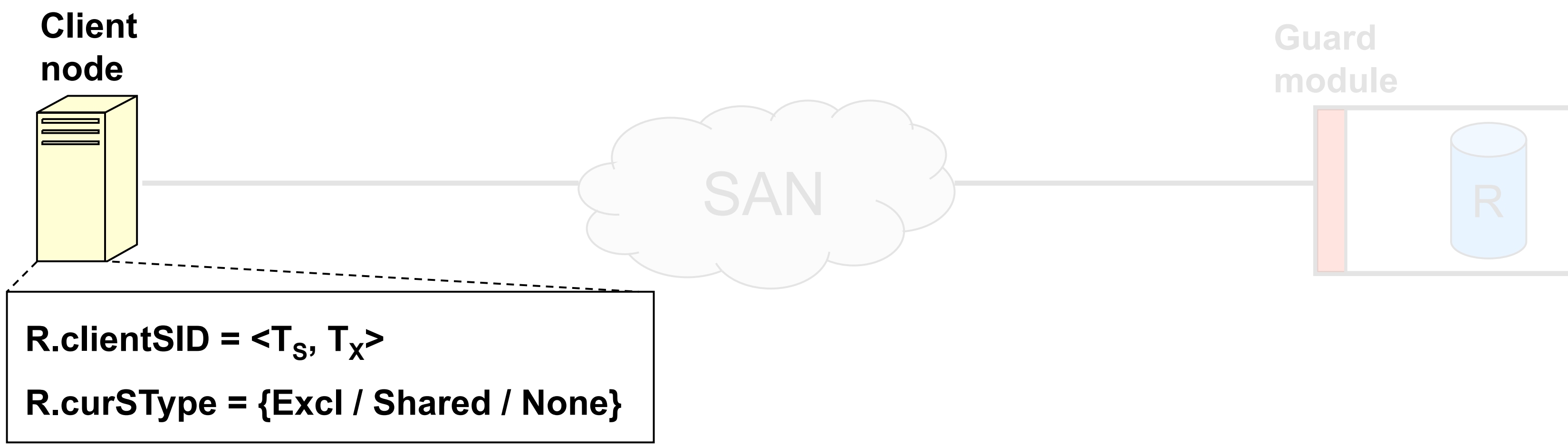
R.clientSID = <T_S, T_X>

R.curSType = {Excl / Shared / None}

Establishing a session to resource R:

```
Lock(R, Shared / Excl) {
    R.curSType ← Shared / Excl
    R.clientSID ← unique session ID
}
```

Enforcing session isolation



Submitting a remote disk command:

<u>Initialize the session annotation:</u>

IF (R.curSType = Excl) {

  updateSID ← R.clientSID

  verifySID ← R.clientSID

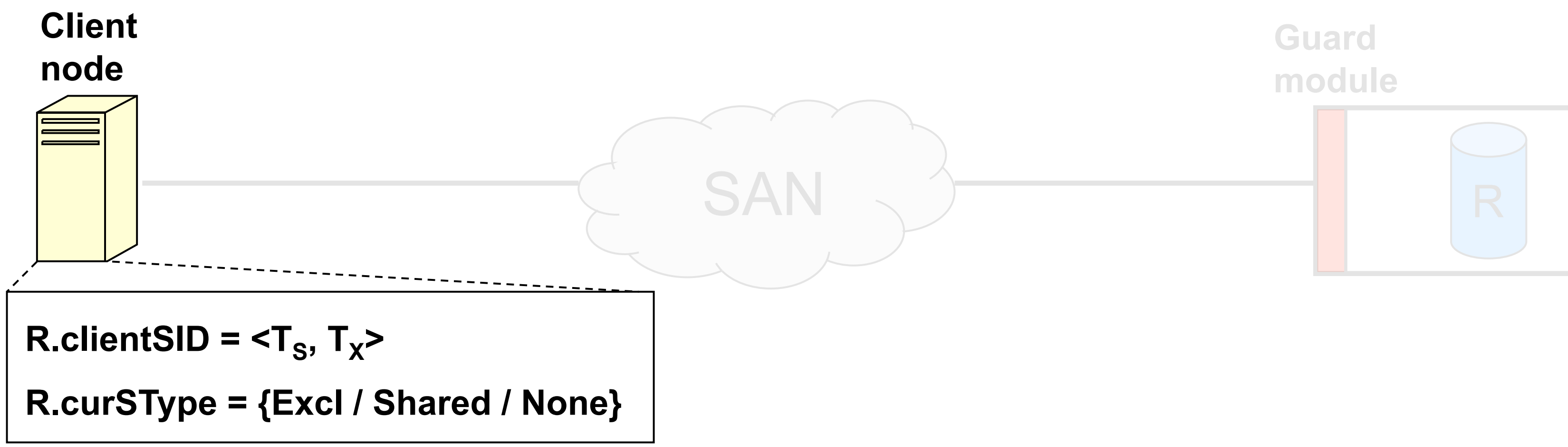
}

command

READ / WRITE (LUN, Offset, Length, …)		
R	**verifySID = <T_s, T_x>**	**updateSID = <T_s, T_x>**

session annotation

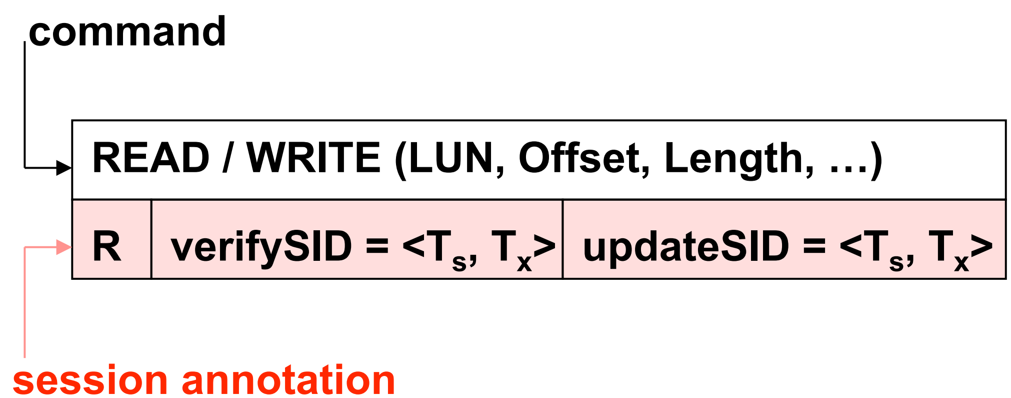
Enforcing session isolation



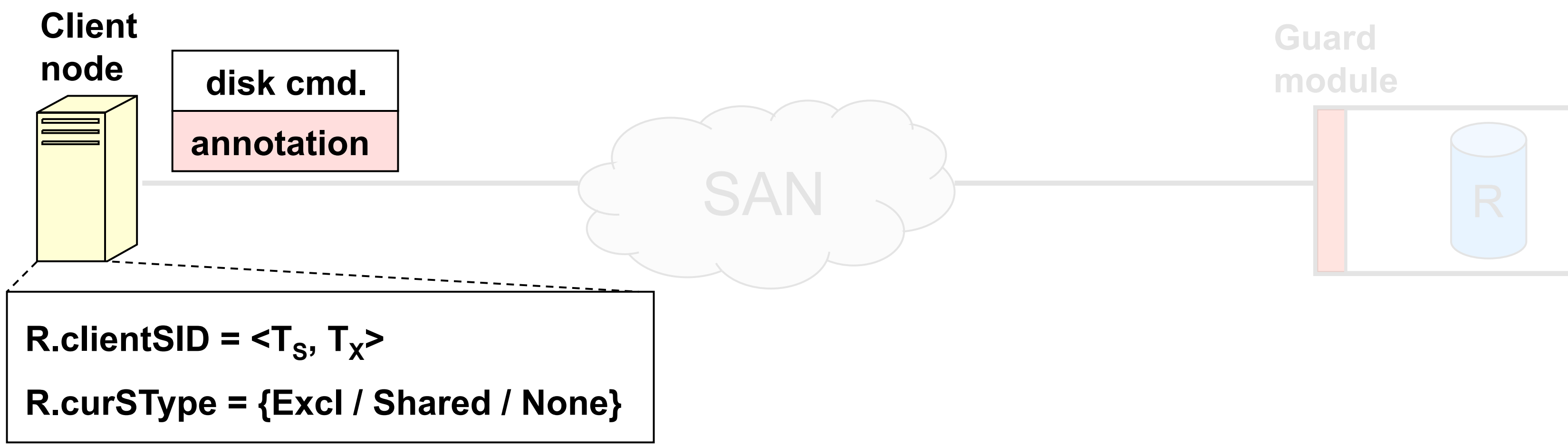
Submitting a remote disk command:

<u>Initialize the session annotation:</u>

IF (R.curSType = Shared) {
  updateSID ← R.clientSID
  verifySID.T_x ← R.clientSID.T_X
  verifySID.T_s ← EMPTY
}



Enforcing session isolation



Submitting a remote disk command:

<u>Initialize the session annotation:</u>

IF (R.curSType = Shared) {
    updateSID ← R.clientSID
    verifySID.T_x ← R.clientSID.T_X
    verifySID.T_s ← EMPTY
}

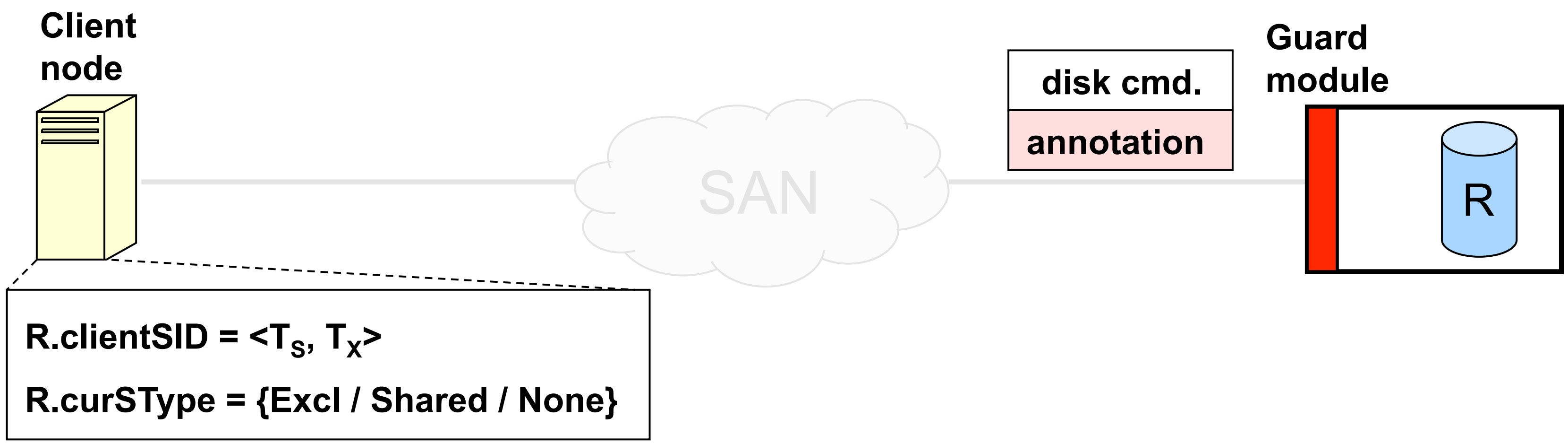
command

READ / WRITE (LUN, Offset, Length, …)

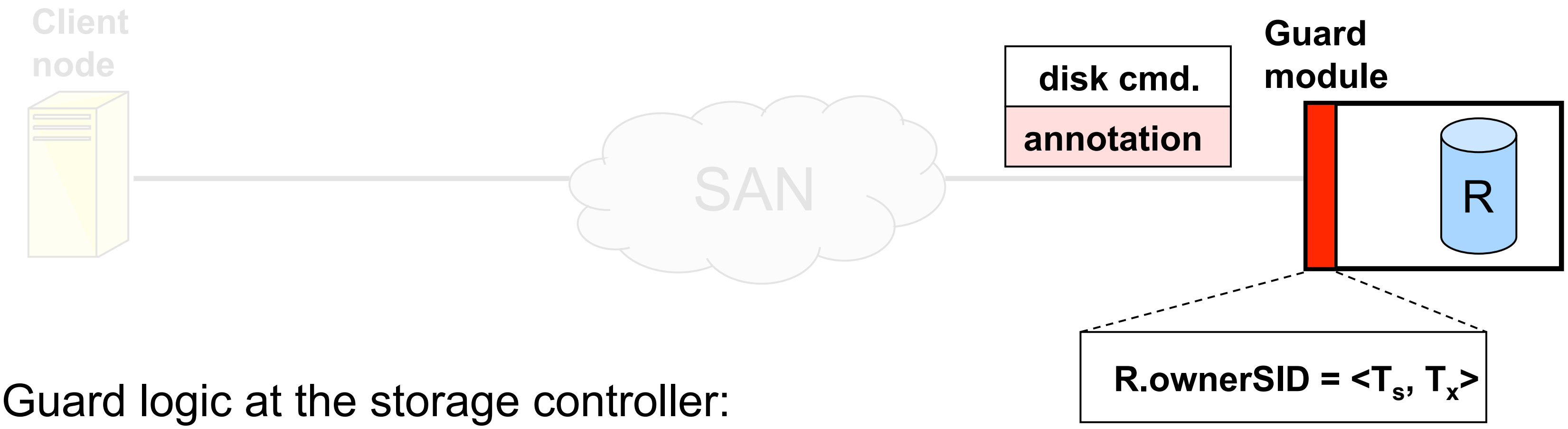
R	**verifySID = $<T_s, T_x>$**	**updateSID = $<T_s, T_x>$**

session annotation

Enforcing session isolation



Enforcing session isolation



Guard logic at the storage controller:

IF (verifySID.T_x < R.ownerSID.T_x)

decision ← REJECT

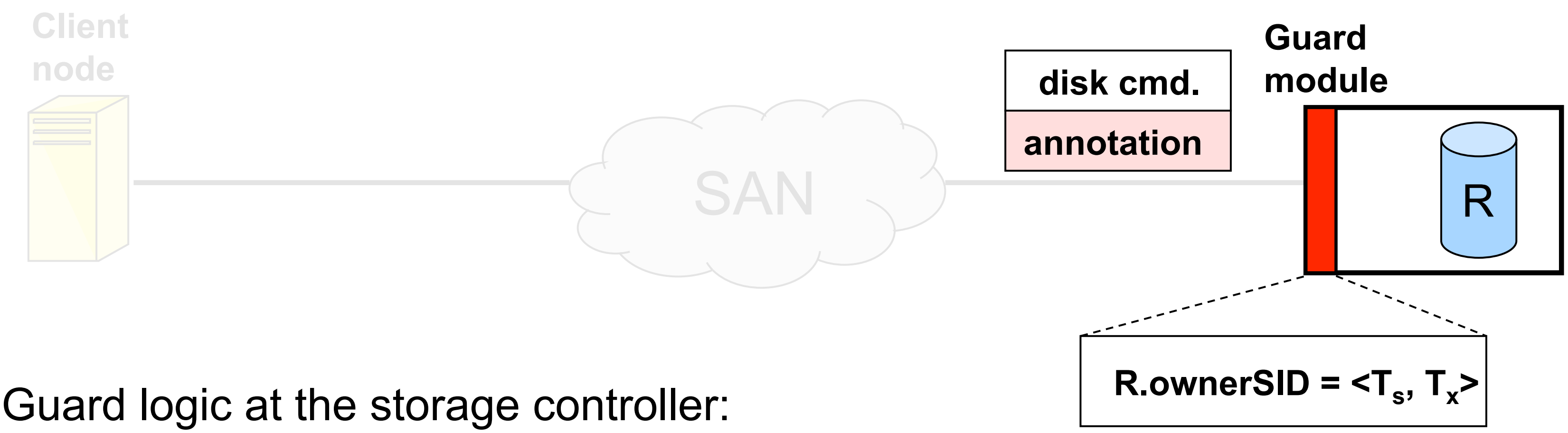
ELSE IF ((verifySID.T_s ≠ EMPTY) AND (verifySID.T_s < R.ownerSID.T_s))

decision ← REJECT

ELSE

decision ← ACCEPT

Enforcing session isolation



Guard logic at the storage controller:

IF (decision = ACCEPT) {

R.ownerSID.T_s ← MAX(R.ownerSID.T_s, updateSID.T_s)

R.ownerSID.T_X ← MAX(R.ownerSID.T_X, updateSID.T_X)

Enqueue and process the command

} ELSE {

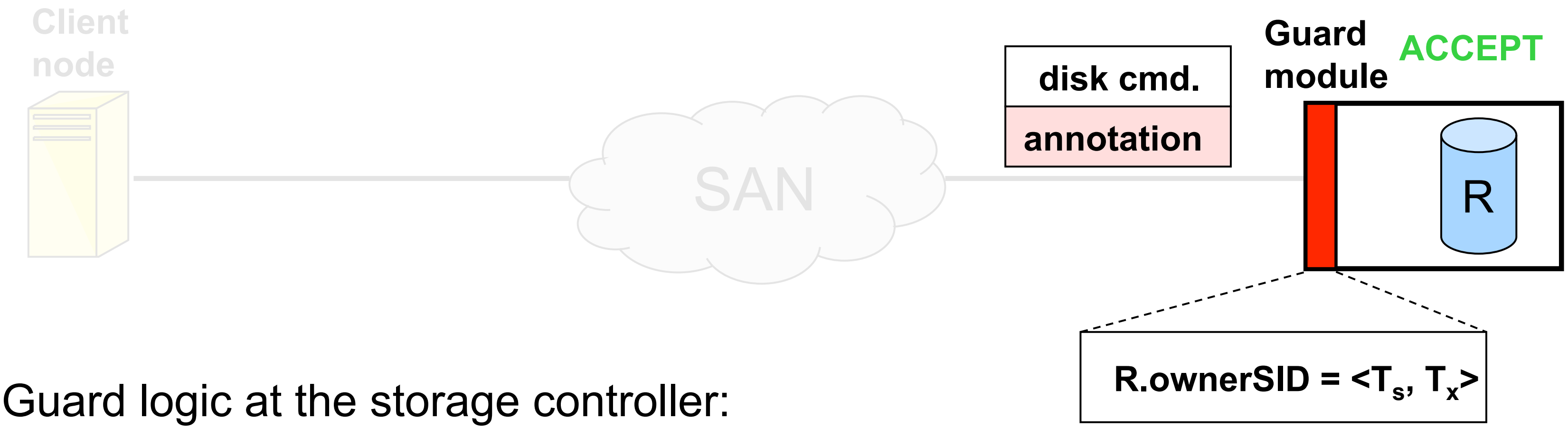
Respond to client with

Status = BADSESSION
R.ownerSID

Drop the command

}

Enforcing session isolation



Guard logic at the storage controller:

IF (decision = ACCEPT) {

R.ownerSID.T_s ← MAX(R.ownerSID.T_s, updateSID.T_s)

R.ownerSID.T_X ← MAX(R.ownerSID.T_X, updateSID.T_X)

Enqueue and process the command

} ELSE {

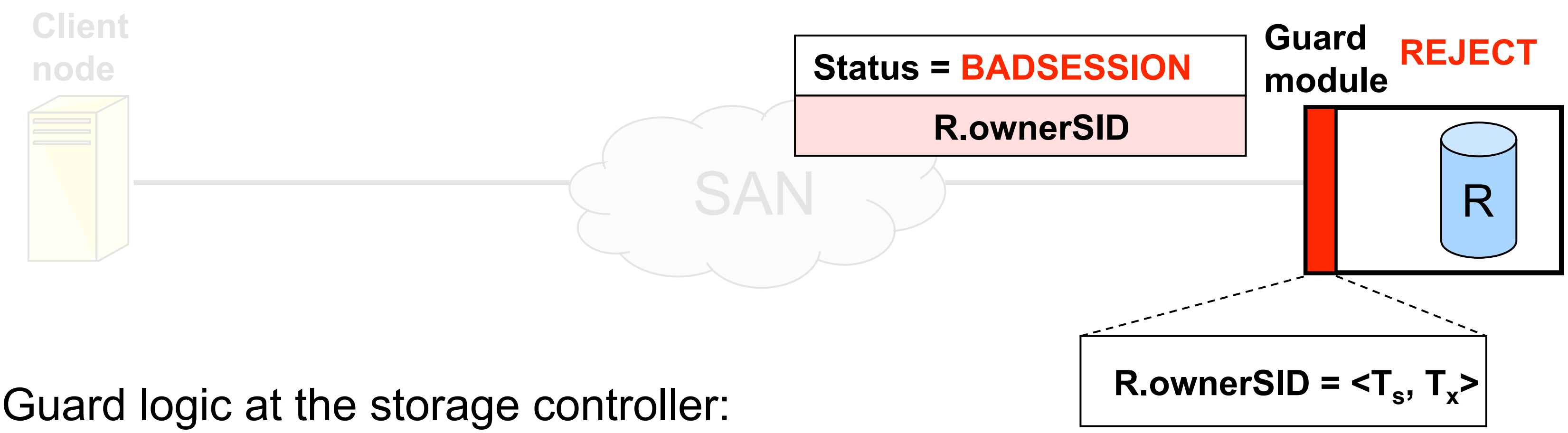
Respond to client with

Status = BADSESSION
R.ownerSID

Drop the command

}

Enforcing session isolation



Guard logic at the storage controller:

IF (decision = ACCEPT) {

  $R.ownerSID.T_s \leftarrow MAX(R.ownerSID.T_s, updateSID.T_s)$

  $R.ownerSID.T_X \leftarrow MAX(R.ownerSID.T_X, updateSID.T_X)$

  <u>Enqueue and process the command</u>

} ELSE {

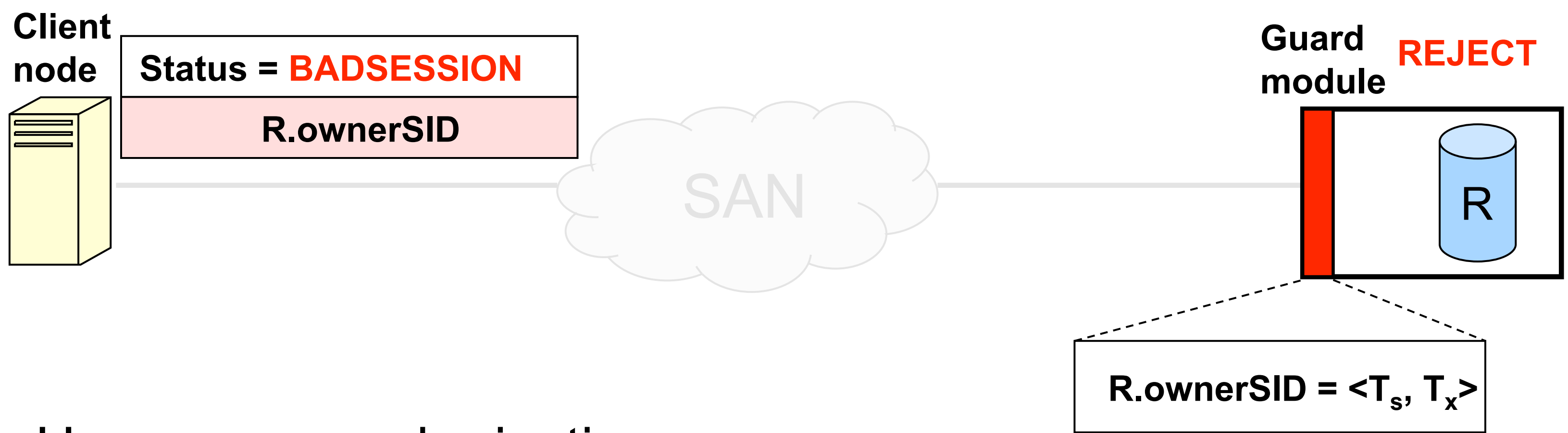
  Respond to client with

Status = BADSESSION
R.ownerSID

  <u>Drop the command</u>

}

Enforcing session isolation



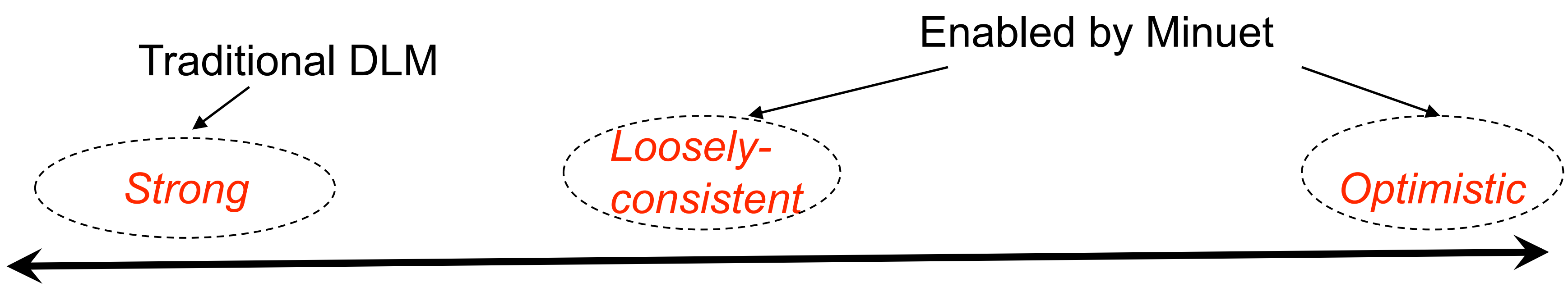
- Upon command rejection:
  - Storage device responds to the client with a special status code (BADSESSION) and the most recent value of R.ownerSID.
  - Application at the client node
    - Observes a failed disk request and forced lock revocation.
    - Re-establishes its session to R under a new SID and retries.

Assignment of session identifiers

- The guard module addresses the safety problems arising from delayed disk request delivery and inconsistent failure observations.

- Enforcing safe ordering of requests at the storage device lessens the demands on the lock service.
  - Lock acquisition state need not be kept consistent at all times.
  - Flexibility in the choice of mechanism for coordination.

Assignment of session identifiers



- SIDs are assigned by a central lock manager.

- Strict serialization of Lock/ Unlock requests.

- Disk command rejection does not occur.

- Performs well under high rates of resource contention.

- Clients choose their SIDs independently and do not coordinate their choices.

- Minimizes latency overhead of synchronization.

- Resilient to network partitions and massive node failures.

- Performs well under low rates of resource contention.

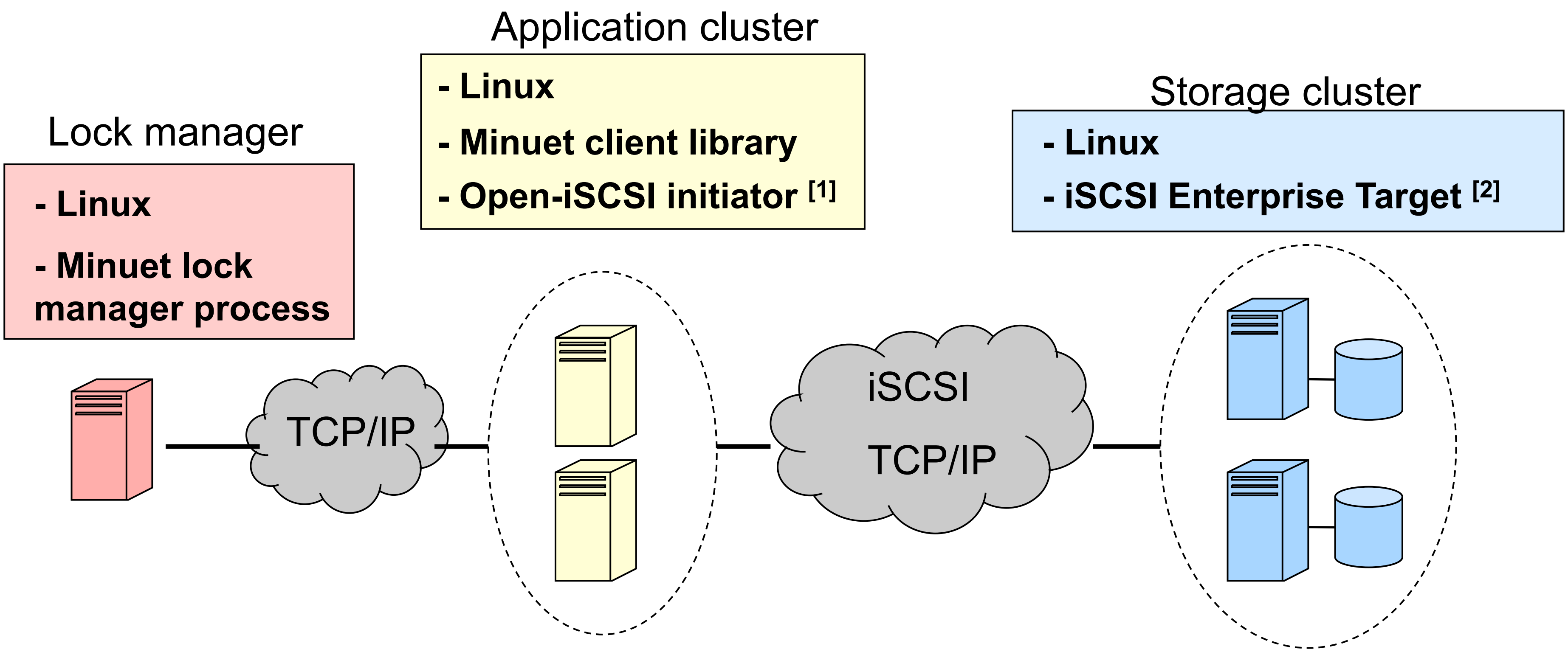
Supporting distributed transactions

- Session isolation provides a building block for more complex and useful semantics.
- Serializable transactions can be supported by extending Minuet with ARIES-style logging and recovery facilities.
- Minuet guard logic:
  - Ensures safe access to the log and the snapshot during recovery.
  - Enables the use of optimistic concurrency control, whereby conflicts are detected and resolved at commit time.

(See paper for details)

Minuet implementation

- We have implemented a proof-of-concept Linux-based prototype and several sample applications.



[1] http://www.open-iscsi.org/ [2] http://iscsitarget.sourceforge.net/

Sample applications

1. Parallel chunkmap (340 LoC)
   - Shared disks store an array of fixed-length data blocks.
   - Client performs a sequence of read-modify-write operations on randomly-selected blocks.
   - Each operation is performed under the protection of an exclusive Minuet lock on the respective block.

Sample applications

2. Parallel key-value store (3400 LoC)
   - B+ Tree on-disk representation.
   - Transactional *Insert*, *Delete*, and *Lookup* operations.
   - Client caches recently accessed tree blocks in local memory.
   - Shared Minuet locks (and content of the block cache) are retained across transactions.
   - With optimistic coordination, stale cache entries are detected and invalidated at transaction commit time.

Emulab deployment and evaluation

- Experimental setup:
  - 32-node application cluster
    - 850MHz Pentium III, 512MB DRAM, 7200 RPM IDE disk
  - 4-node storage cluster
    - 3.0GHz 64-bit Xeon, 2GB DRAM, 10K RPM SCSI disk
  - 3 Minuet lock manager nodes
    - 850MHz Pentium III, 512MB DRAM, 7200 RPM IDE disk

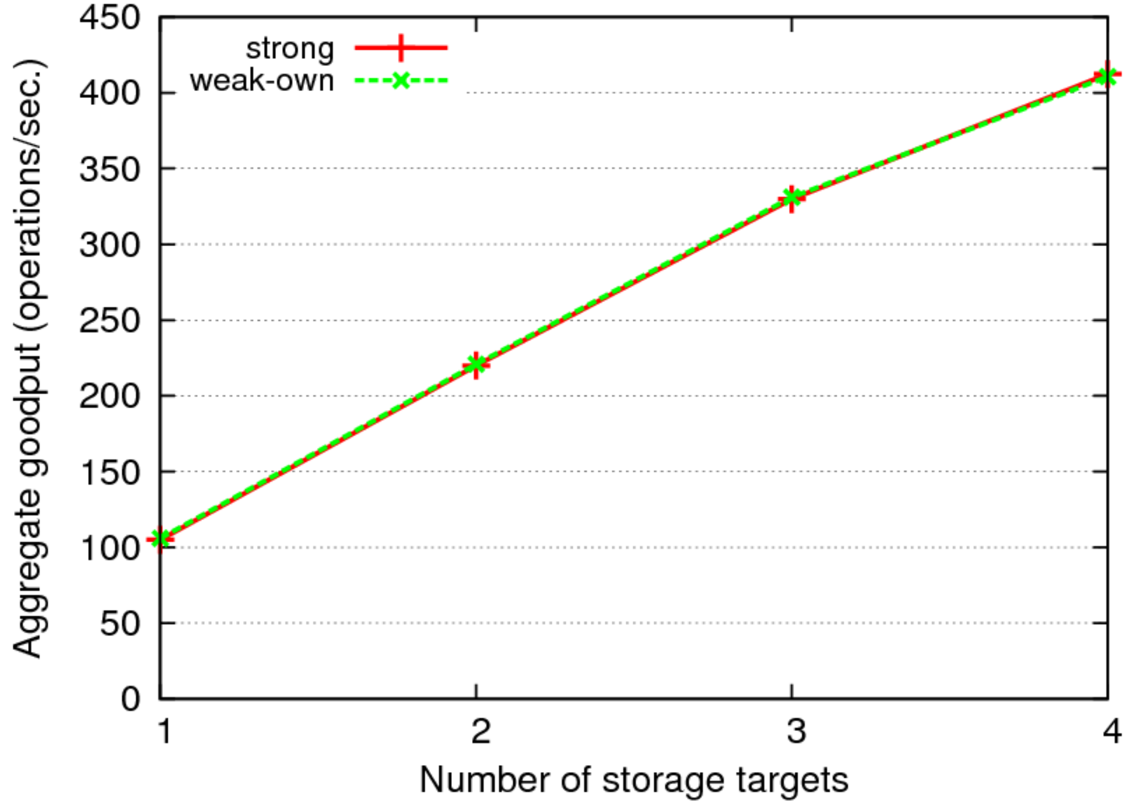
  - 100Mbps Ethernet

Emulab deployment and evaluation

- Measure application performance with two methods of concurrency control:
  - *Strong*
    - Application clients coordinate through one Minuet lock manager process that runs on a dedicated node.
    - “Traditional” distributed locking.
  - *Weak-own*
    - Each client process obtains locks from a local Minuet lock manager instance.
    - No direct inter-client coordination.
    - “Optimistic” technique enabled by our approach.

Parallel chunkmap: Uniform workload

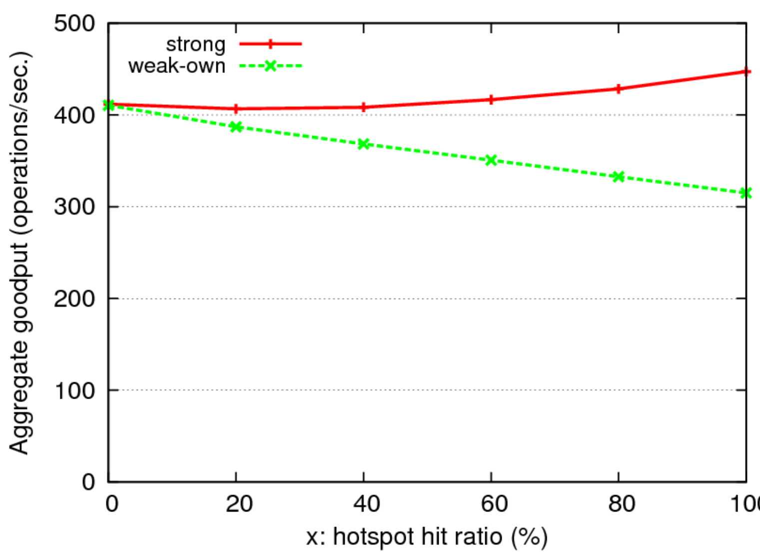
- 250,000 data chunks striped across [1-4] storage nodes.
- 8KB chunk size, 32 chunkmap client nodes
- Uniform workload:
  clients select chunks
  uniformly at random.

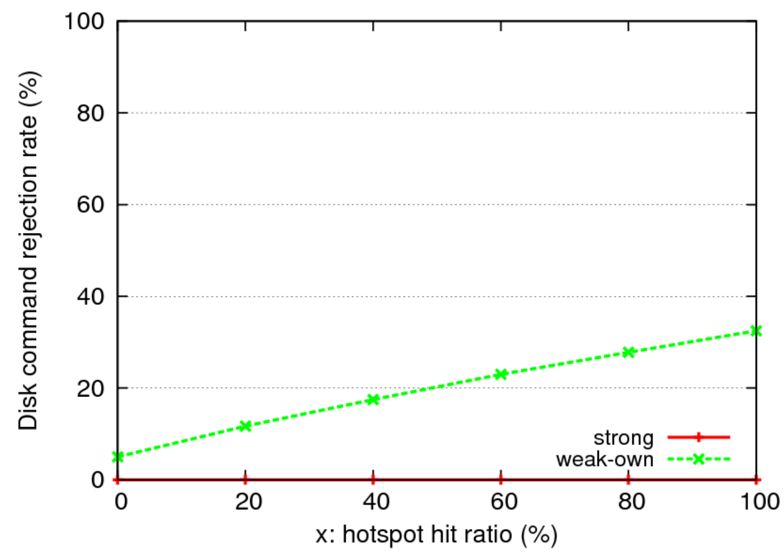


Parallel chunkmap: Hotspot workload

- 250,000 data chunks striped across 4 storage nodes.
- 8KB chunk size, 32 chunkmap client nodes
- Hotspot(x) workload: x% of operations touch a “hotspot” region of the chunkmap.

  Hotspot size = 0.1% = 2MB.



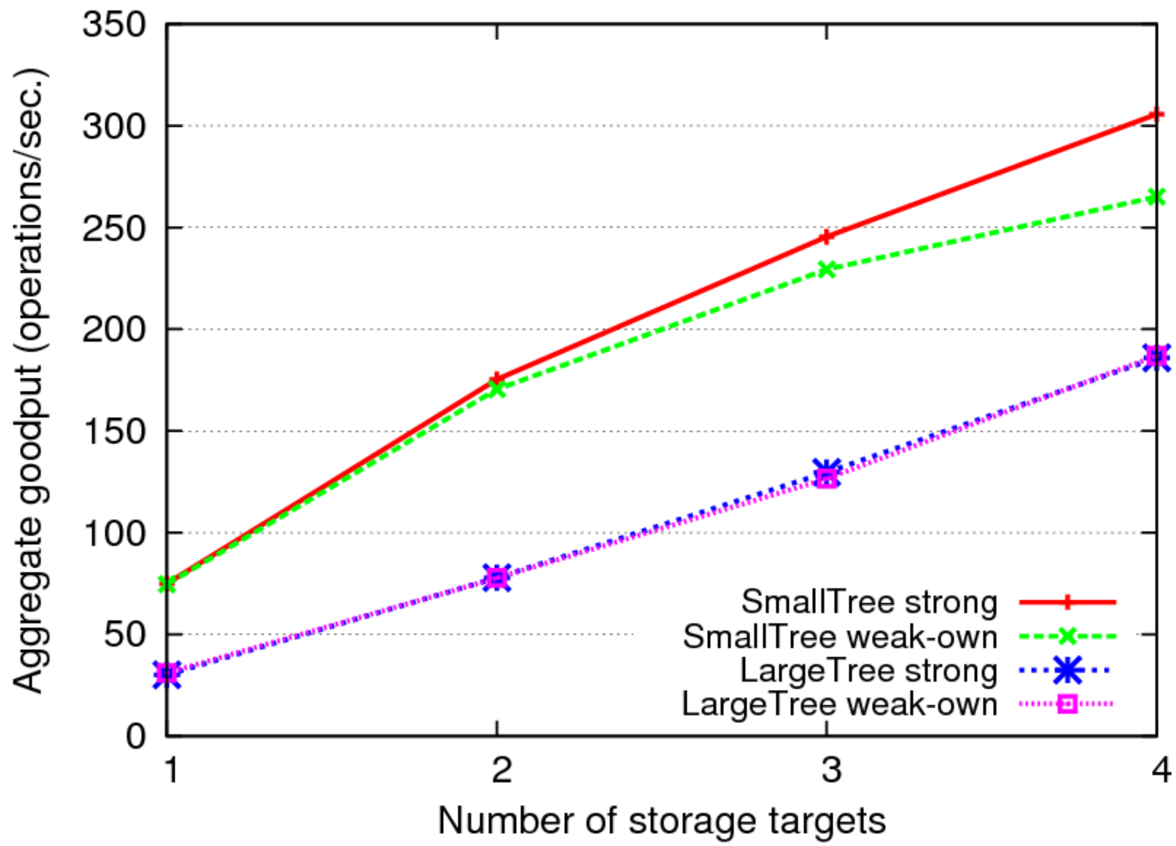


Experiment 2: Parallel key-value store

	SmallTree	LargeTree
Block size	8KB	8KB
Fanout	150	150
Depth	3 levels	4 levels
Initial leaf occupancy	50%	50%
Number of keys	187,500	18,750,000
Total dataset size	20MB	2GB

Experiment 2: Parallel key-value store

- [1-4] storage nodes.
- 32 application client nodes.
- Each client performs a series of random key-value insertions.



Challenges

- Practical feasibility and barriers to adoption
  - Extending storage arrays with guard logic
- Medatada storage overhead (table of *ownerSIDs*).
- SAN bandwidth overhead due to session annotations
- Changes to the programming model
  - Dealing with I/O command rejection and forced lock revocations

Related Work

- Optimistic concurrency control (OCC) in database management systems.
- Device-based locking for shared-disk environments (*Dlocks*, *Device Memory Export Protocol).*
- Storage protocol mechanisms for failure fencing (*SCSI-3 Persistent Reserve*).
- New synchronization primitives for datacenter applications (*Chubby*, *Zookeeper*).

Summary

- Minuet is a new synchronization primitive for clustered shared-disk applications and middleware.
- Augments shared storage devices with *guard logic*.
- Enables the use of OCC as an alternative to conservative locking.
- Guarantees data safety in the face of arbitrary asynchrony.
  - Unbounded network transfer delays
  - Unbounded clock drift rates
- Improves application availability.
  - Resilience to large-scale node failures and network partitions

Thank you !

Backup Slides

Related Work

- Optimistic concurrency control (OCC)
  - Well-known technique from the database field.
  - Minuet enables the use of OCC in clustered SAN applications as an alternative to “conservative” distributed locking.

Related Work

- Device-based synchronization (*Dlocks*, *Device Memory Export Protocol)*
  - Minuet revisits this idea from a different angle; provides a more general primitive that supports both OCC and traditional locking.
  - We extend storage devices with *guard logic* – a minimal functional component that enables both approaches.

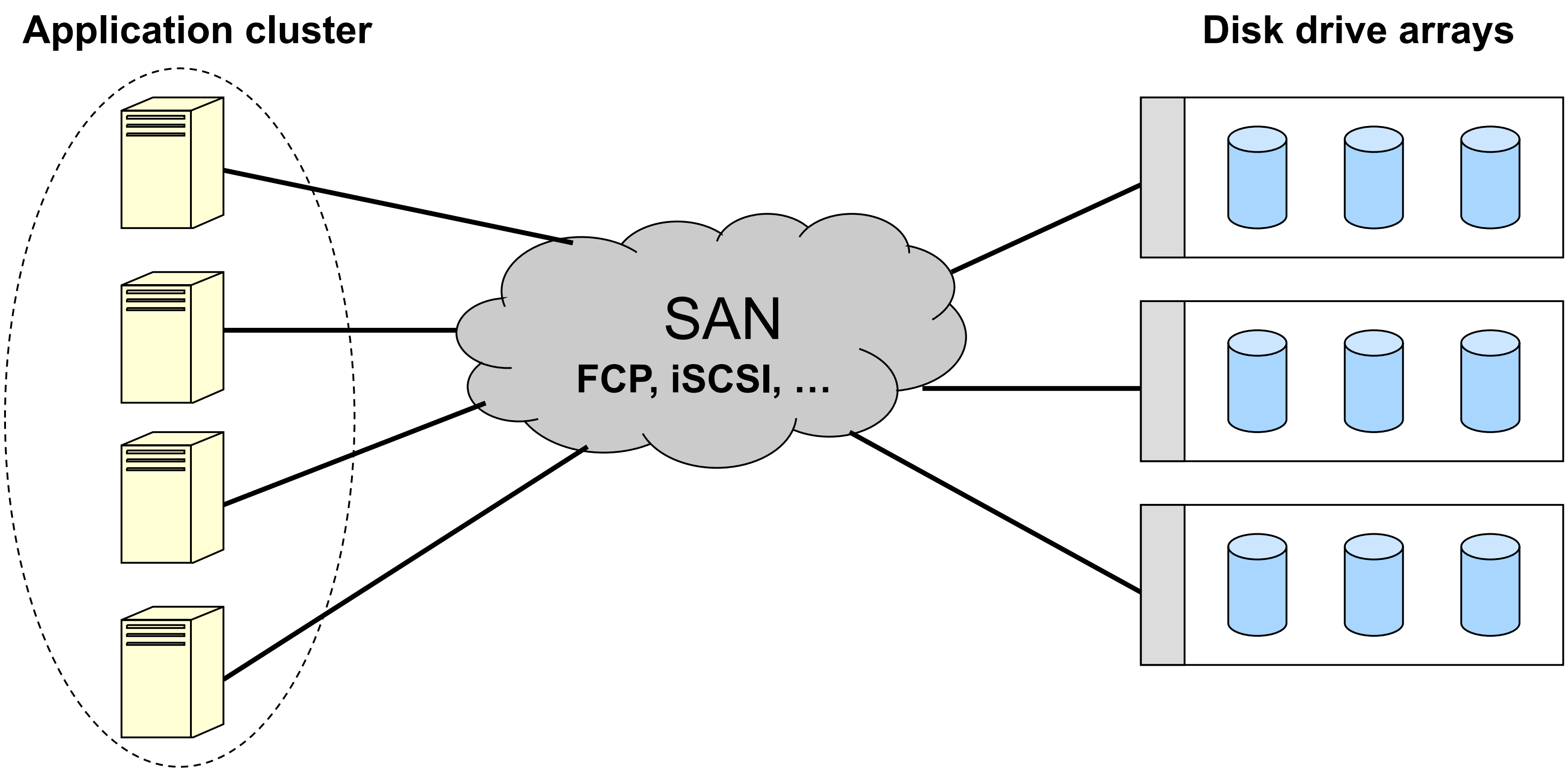
Related Work

- Storage protocol mechanisms for failure fencing (*SCSI-3 Persistent Reserve*)
  - PR prevents out-of-order delivery of delayed disk commands from (suspected) faulty nodes.
  - Ensures safety but not availability in a partitioned network; Minuet provides both.

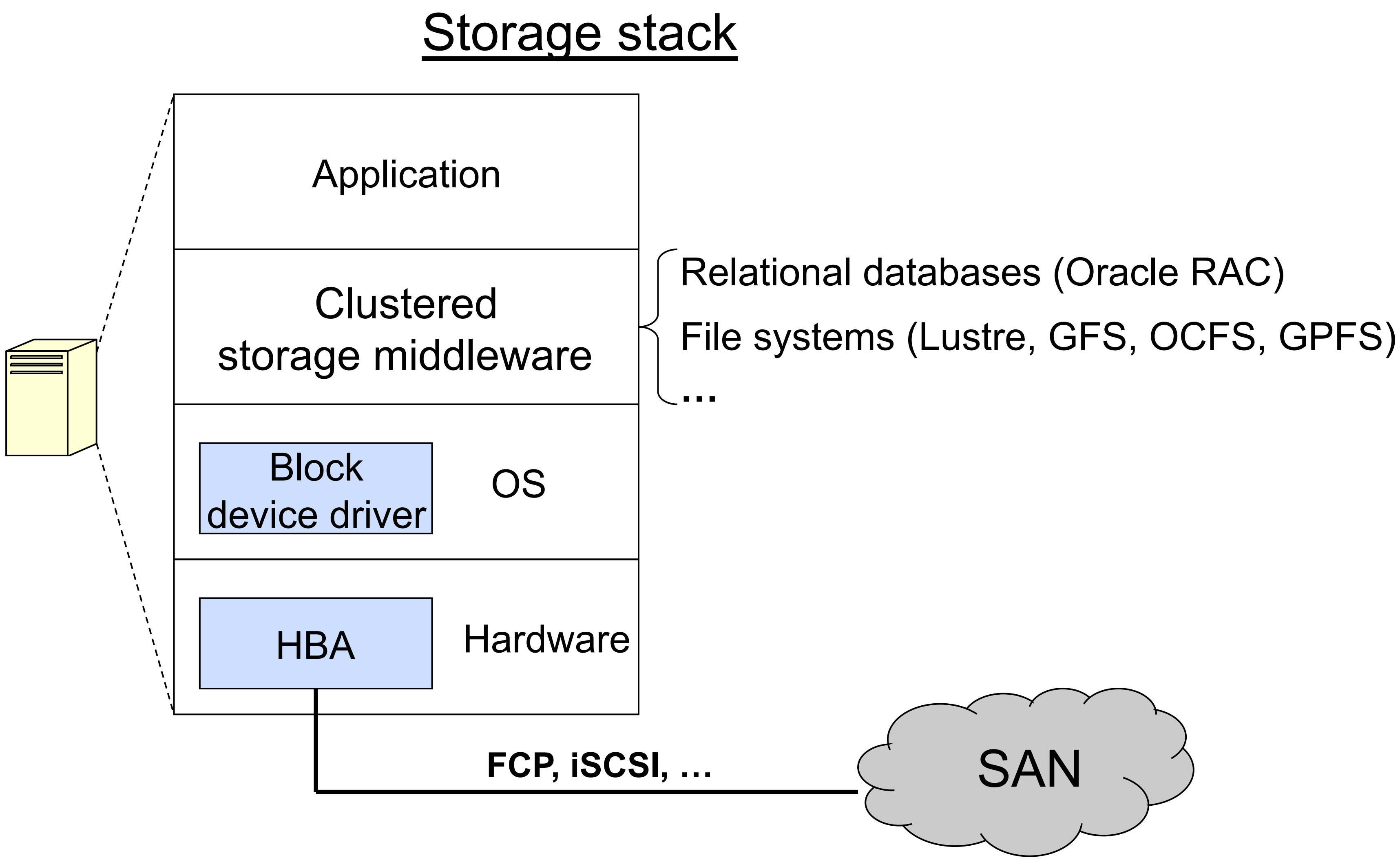
Related Work

- New synchronization primitives for datacenter applications (*Chubby*, *Zookeeper*).
  - Minuet focuses on *fine-grained* synchronization for clustered SAN applications.
  - Minuet's *session annotations* are conceptually analogous to Chubby's *lock sequencers.*
    - We extend this mechanism to shared-exclusive locking.
    - Given the ability to reject out-of-order requests at the destination, global consistency on the state of locks and use of an agreement protocol may be more than necessary.
    - Minuet attains improved availability by relaxing these consistency constraints.

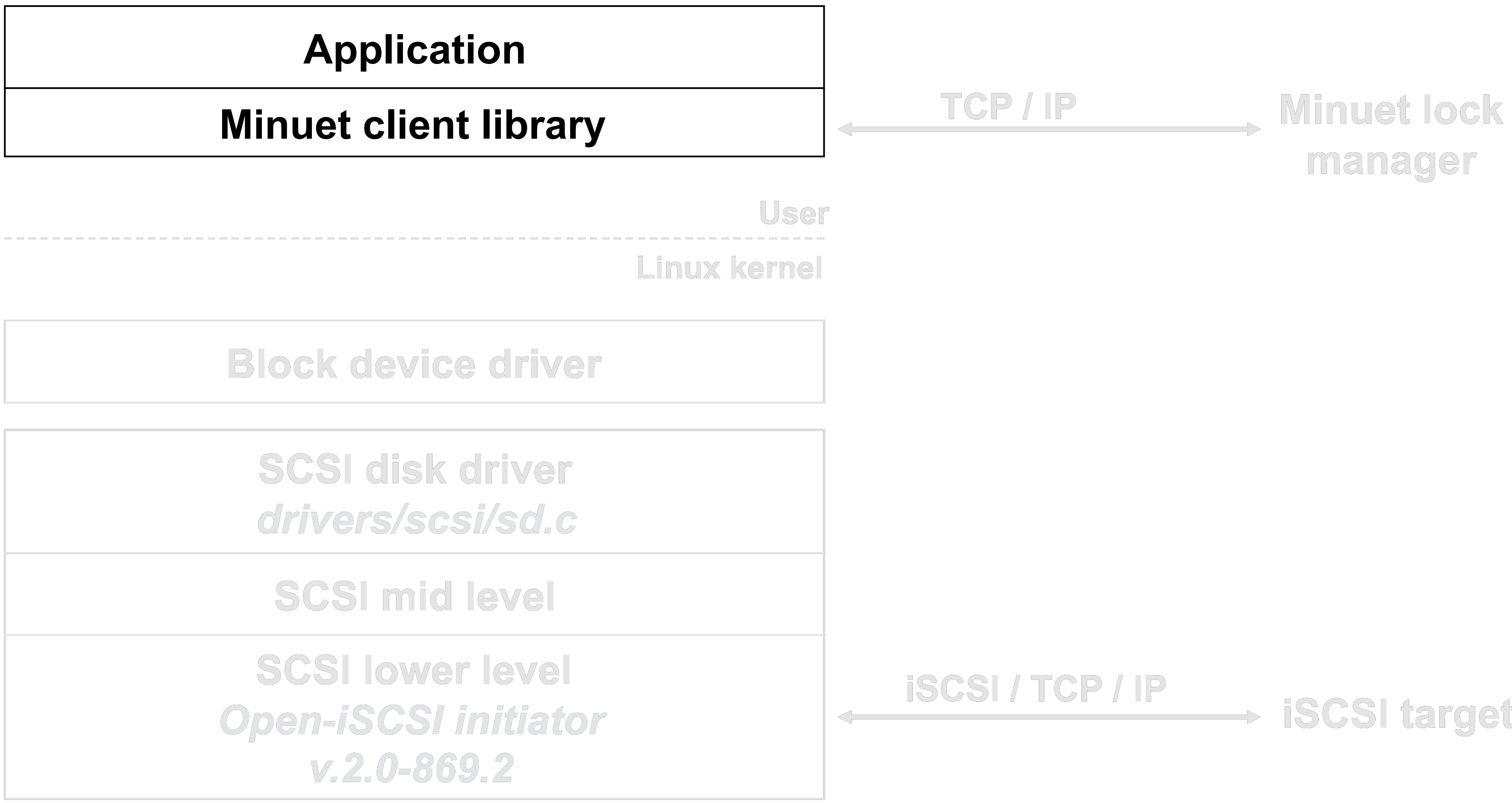
Clustered SAN applications and services



Clustered SAN applications and services



Minuet implementation: application node



Minuet API

Lock service

- MinuetUpgradeLock(resource_id, lock_mode);
- MinuetDowngradeLock(resource_id, lock_mode);

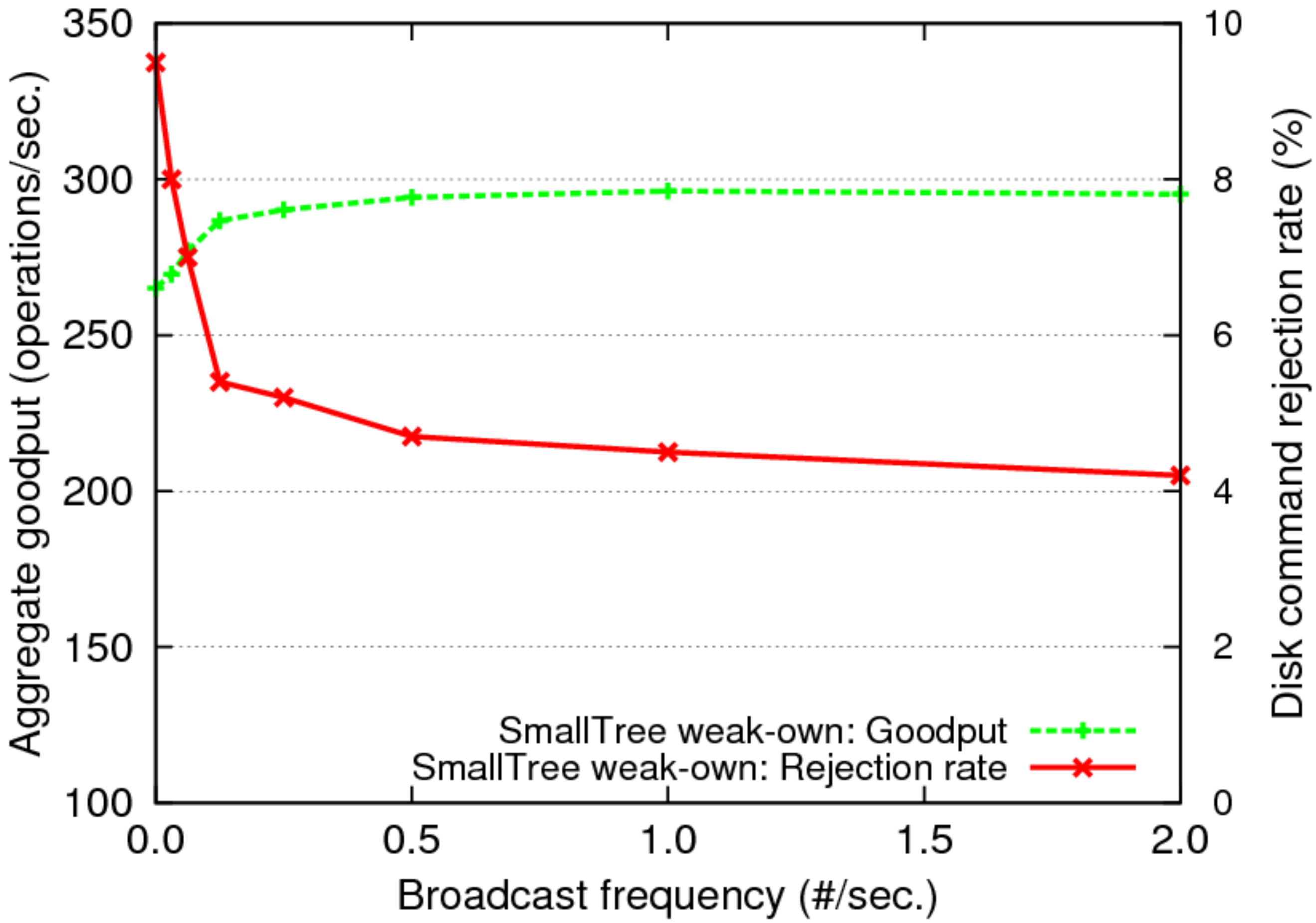
Remote disk I/O

- MinueDiskRead(lun_id, resource_id, start_sector, length, data_buf);
- MinueDiskWrite(lun_id, resource_id, start_sector, length, data_buf);

Transaction service

- MinuetXactBegin();
- MinuetXactLogUpdate(lun_id, resource_id, start_sector, length, data_buf);
- MinuetXactCommit(readset_resource_ids[], writeset_resource_ids[]);
- MinuetXactAbort();
- MinuetXactMarkSynched();

Experiment 2: B+ Tree



Supporting serializable transactions

- Five stages of a transaction (*T*): (see paper for details)

1) READ
   - Acquire shared Minuet locks on *T.ReadSet*; Read these resources from shared disk.
2) UPDATE
   - Acquire exclusive Minuet locks on the elements of *T.WriteSet*; Apply updates locally; Append description of updates to the log.
3) PREPARE
   - Contact the storage devices to verify validity of all sessions in *T* and lock *T.WriteSet* in preparation for commit.
4) COMMIT
   - Force-append a *Commit* record to the log.
5) SYNC (proceeds asynchronously)
   - Flush all updates to shared disks and unlock *T.WriteSet*.

Minuet implementation

- Extensions to the storage stack:
  - Open-iSCSI Initiator on application nodes:
    - Minuet session annotations are attached to outbound command PDUs using the Additional Header Segment (AHS) protocol feature of iSCSI.
  - iSCSI Enterprise Target on storage nodes:
    - Guard logic (350 LoC; 2% increase in complexity).
    - ownerSIDs are maintained in main memory using a hash table.
    - Command rejection is signaled to the initiator via a Reject PDU.