Case Study 1: Consensus in Raft

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

What is Raft?

- A library that uses a leader-based consensus scheme to implement fault-tolerant state machine replication

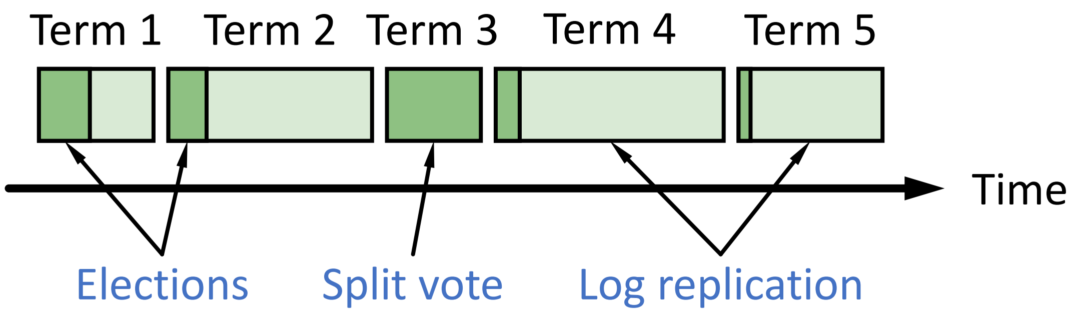
- Keys components
  - Leader election: elects one leader at a time — Lab 3A
  - Log replication: leader broadcasts messages to replicas in order — Lab 3B
  - Crash recovery: handles crashed replicas — Lab 3C
  - Log compaction: discards obsolete log entries — n/a
  - Client interaction: ensures exactly-once semantics — Lab 4

- See animation: https://thesecretlivesofdata.com/raft/

Leader Election

Terms (aka Epochs)

- Raft divides time into terms
- Each term starts with leader election
  - If election fails, a term has no leader (e.g., Term 3)
  - Otherwise, a term has one leader that performs log replication
- Each replica maintains latest known term value
  - Updates value on receiving request/response with higher value
  - Rejects requests from previous terms, responds with current term

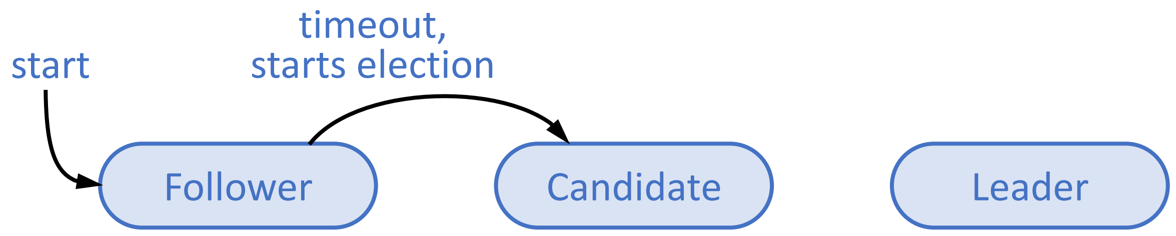


Replica states

- At any given time, each replica is in one three states:
  - Leader: handles all client interactions, performs log replication
  - Follower: receives messages from leader, completely passive
  - Candidate: starts election to become new leader

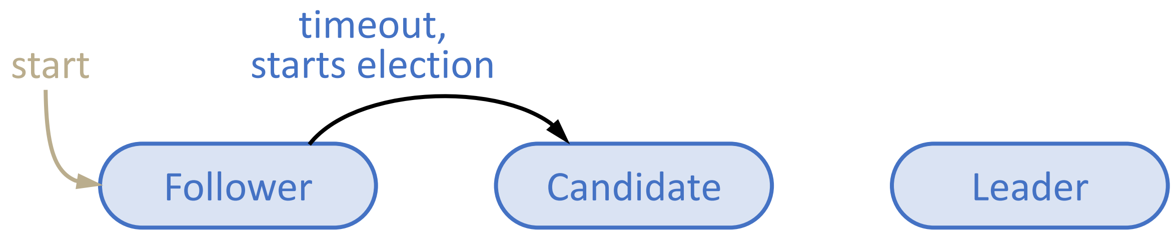
Starting an election

- All replicas start as followers
- After leader is elected, it sends periodic heartbeats to maintain authority over followers



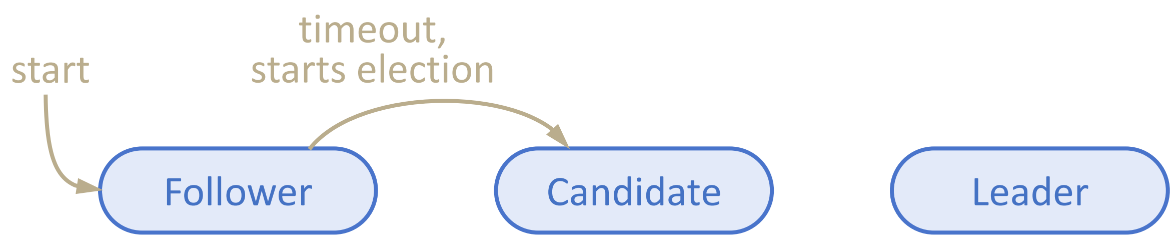
Starting an election

- All replicas start as followers
- After leader is elected, it sends periodic heartbeats to maintain authority over followers
- If a follower doesn't receive a heartbeat within an election timeout, it assumes leader has crashed
- Starts election by incrementing current term, changing to candidate state, voting for self



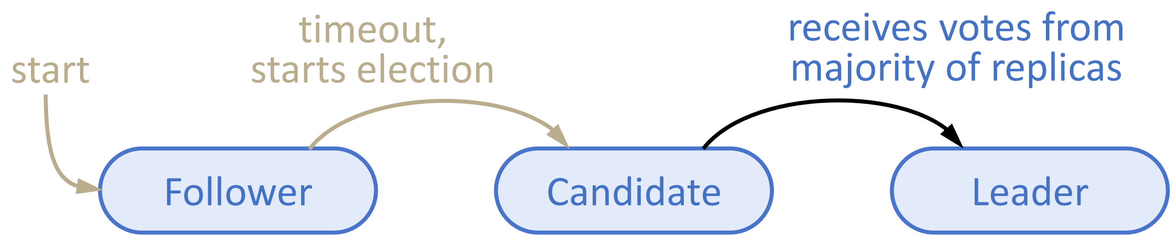
Election

- Candidate sends RequestVote to all other replicas



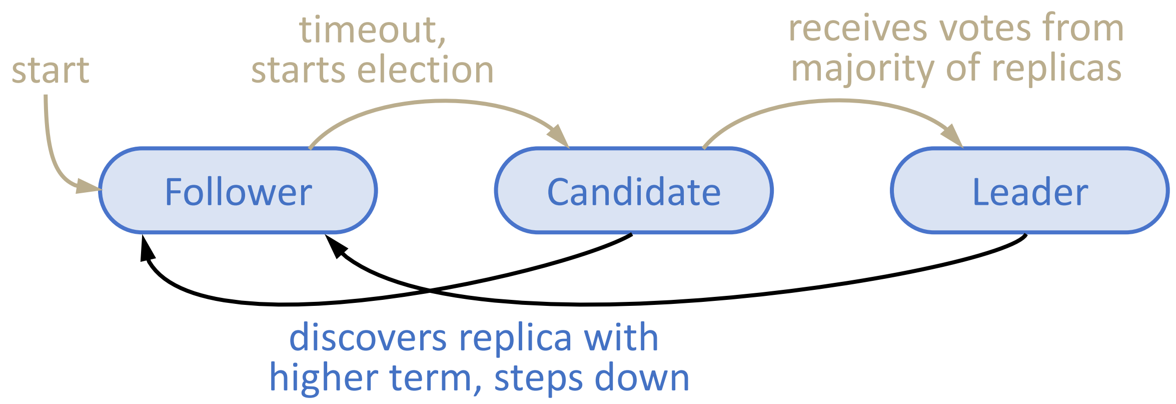
Election

- Candidate sends RequestVote to all other replicas
    - Receives votes from majority of replicas:
        - Becomes leader
        - Sends heartbeats to tell all other replicas it is new leader



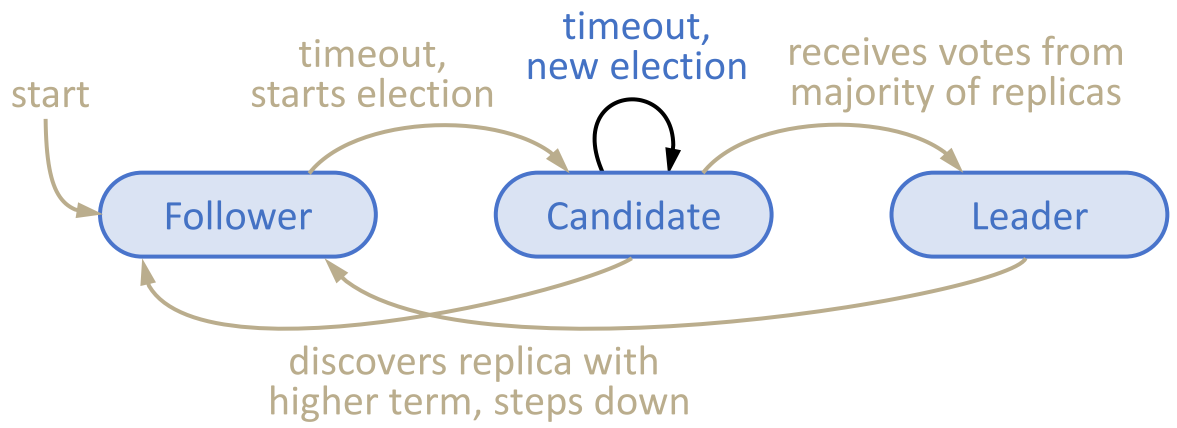
Election

- Candidate sends RequestVote to all other replicas
  - Receives votes from majority of replicas:
    - Becomes leader
    - Sends heartbeats to tell all other replicas it is new leader
  - Receives heartbeat from valid leader (with same/higher term):
    - Returns to follower state


start
timeout, starts election
receives votes from majority of replicas
Follower
Candidate
Leader
discovers replica with higher term, steps down

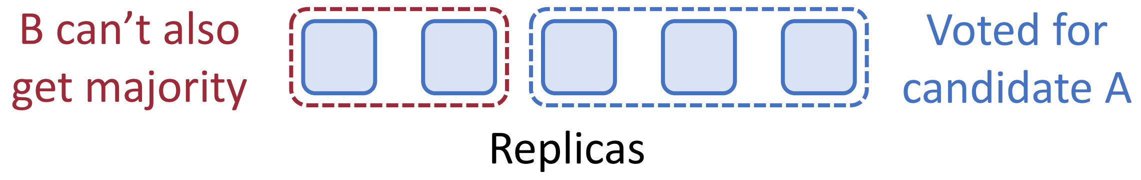
Election

- Candidate sends RequestVote to all other replicas:
    - Receives votes from majority of replicas:
        - Becomes leader
        - Sends heartbeats to tell all other replicas it is new leader
    - Receives heartbeat from valid leader (with same/higher term):
        - Returns to follower state
    - Election timeout elapses (election failed):
        - Increments term, starts new election


start
timeout, starts election
timeout, new election
receives votes from majority of replicas
Follower
Candidate
Leader
discovers replica with higher term, steps down

Safety

- Safety: allow at most one winner per term
- Each replica votes only once per term
  - Votes for first candidate that asks
  - Vote is stored on disk durably (ensures safety under crashes)
- Two candidates can't get majorities in same term



- What if previous leader isn't aware of new leader?
  - It will either learn about a new term and step down,
    or not be able to perform log replication (we will see later)

Liveness

- Liveness: some candidate eventually wins
- Suppose followers/candidates have same election timeout, could there be a problem?
- Followers/candidates choose election timeout randomly
  - Randomness reduces chance of split vote by breaking symmetry
  - One follower usually initiates, wins election before others start

Choosing election timeout

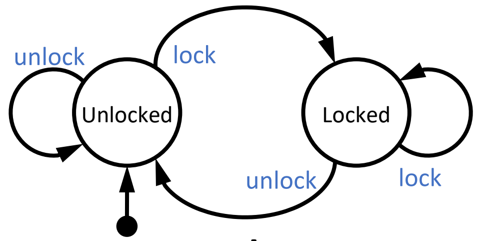
- Choice of election timeout affects liveness
    - Should be short to reduce unavailability
        - After leader crashes, system becomes unavailable for election timeout
    - Should be at least a few multiple of heartbeat intervals
        - Avoids unneeded election if a heartbeat from leader is lost
    - Random part should be several network round-trip times
        - A candidate can win an election before others start
- Election timeout chosen randomly between 150-300 ms
    - Assumes heartbeat interval in the 10 ms range

Log Replication

Overview of log replication

- Raft uses log replication to broadcast clients' operations in FIFO-total order
- A client issues an operation at the leader
- Leader logs the operation, broadcasts them to followers
- Followers log the operation, respond to the leader
- Raft ensures that the logs at the replicas remain consistent
- Operations are executed at replicas in log order, which ensures total order

Replicated lock service

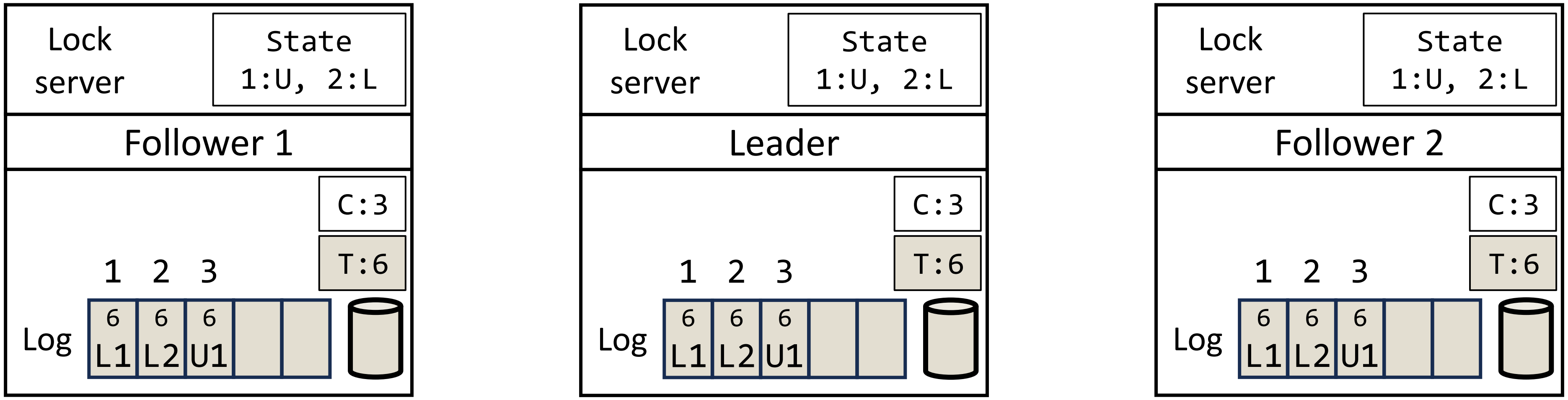


- Let's use a lock service to show how log replication works
  - Lock service has two operations: lock, unlock
  - Lock service maintains unlocked/locked state per lock
- When a client sends an operation to lock service, leader replica will invoke Raft to replicate the operation
- Raft will use log replication to broadcast operation to all replicas

Replicated lock service

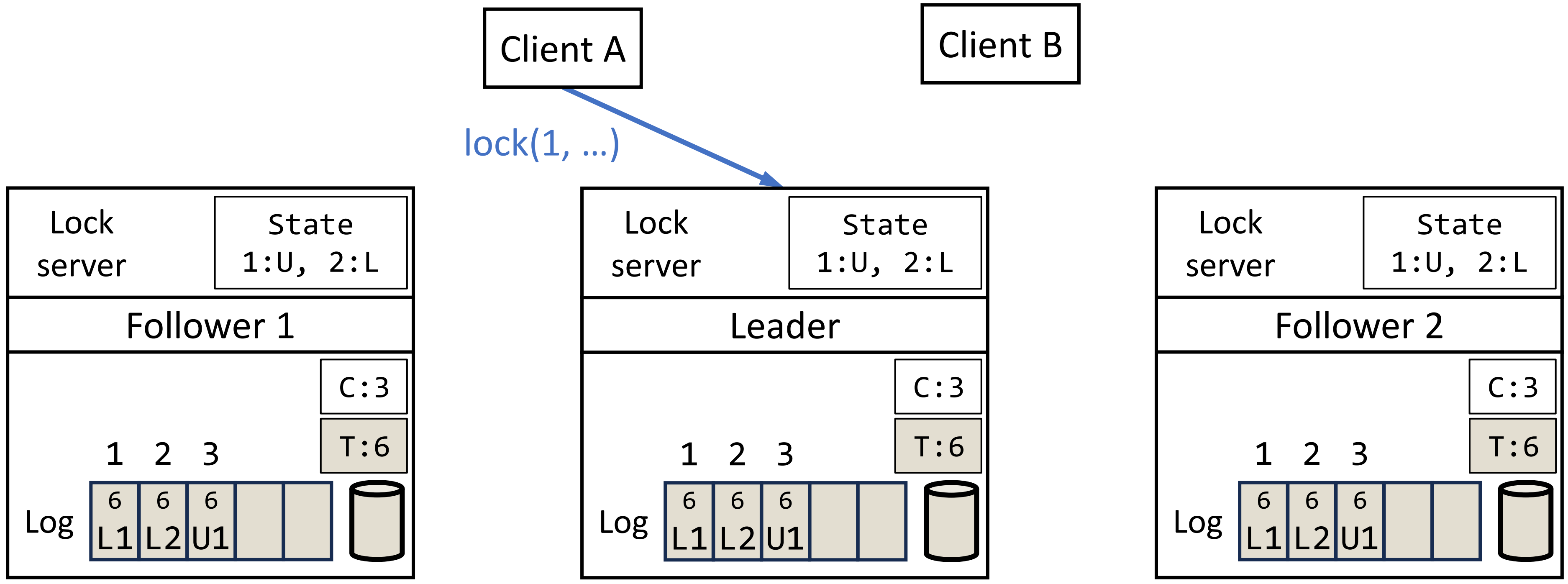
- Assume two locks 1, 2: Currently 1 is Unlocked, 2 is Locked
- Assume leader is already elected
- At each replica, Raft maintains:

In memory	Highest log entry known to be committed – C : 3
In non-volatile storage, e.g., disk, SSD, cached in memory	Log containing history of <operations, term number>, Latest known term – T : 6



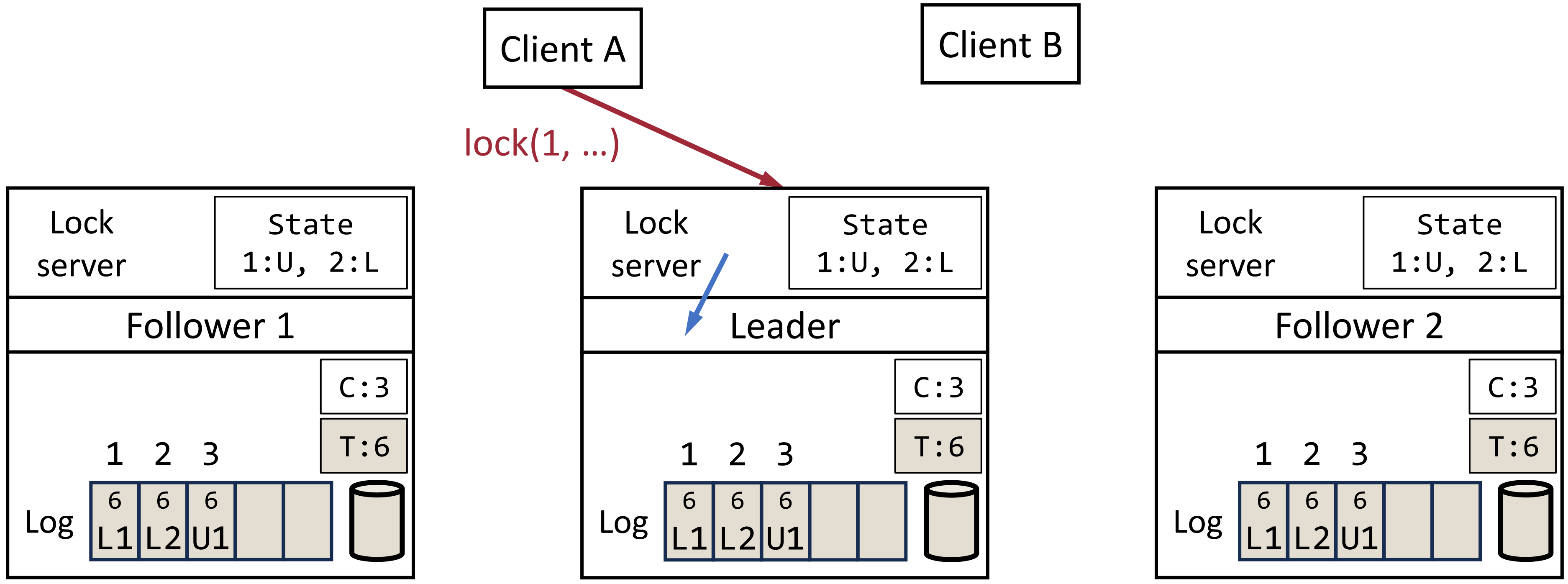
Replicated lock service

- Client invokes lock operation at leader replica



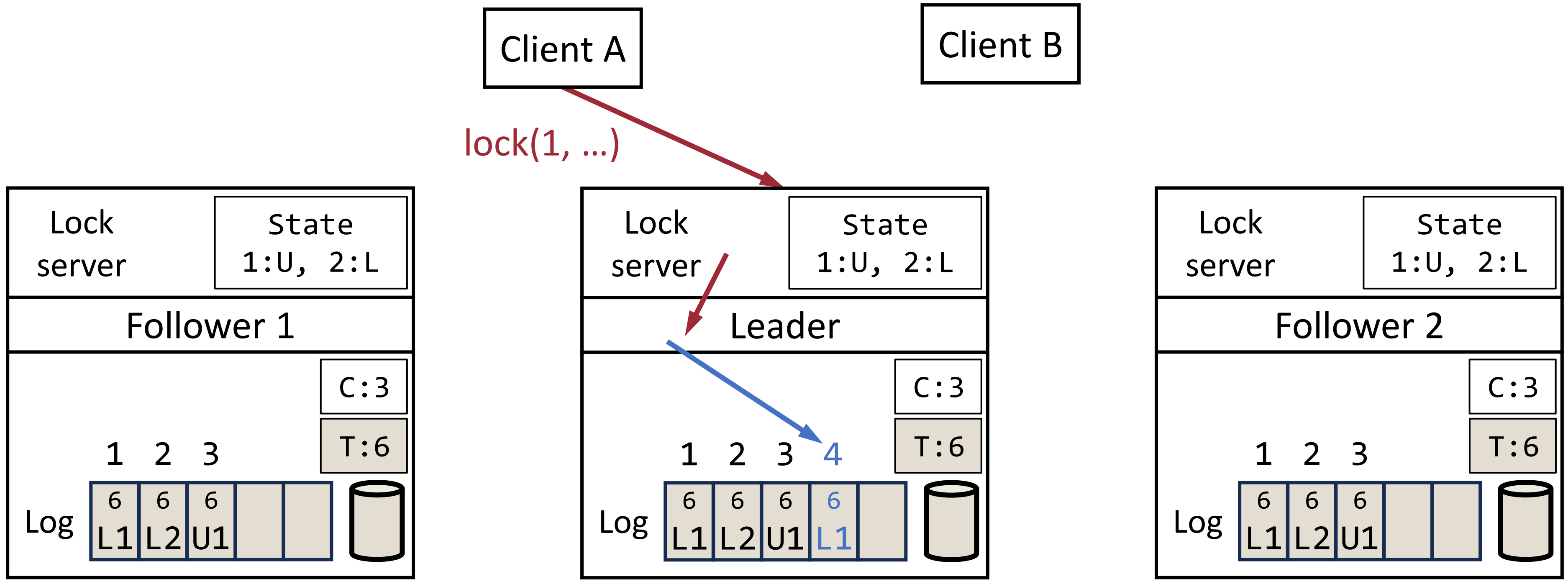
Replicated lock service

- Lock server forwards operation to Raft library



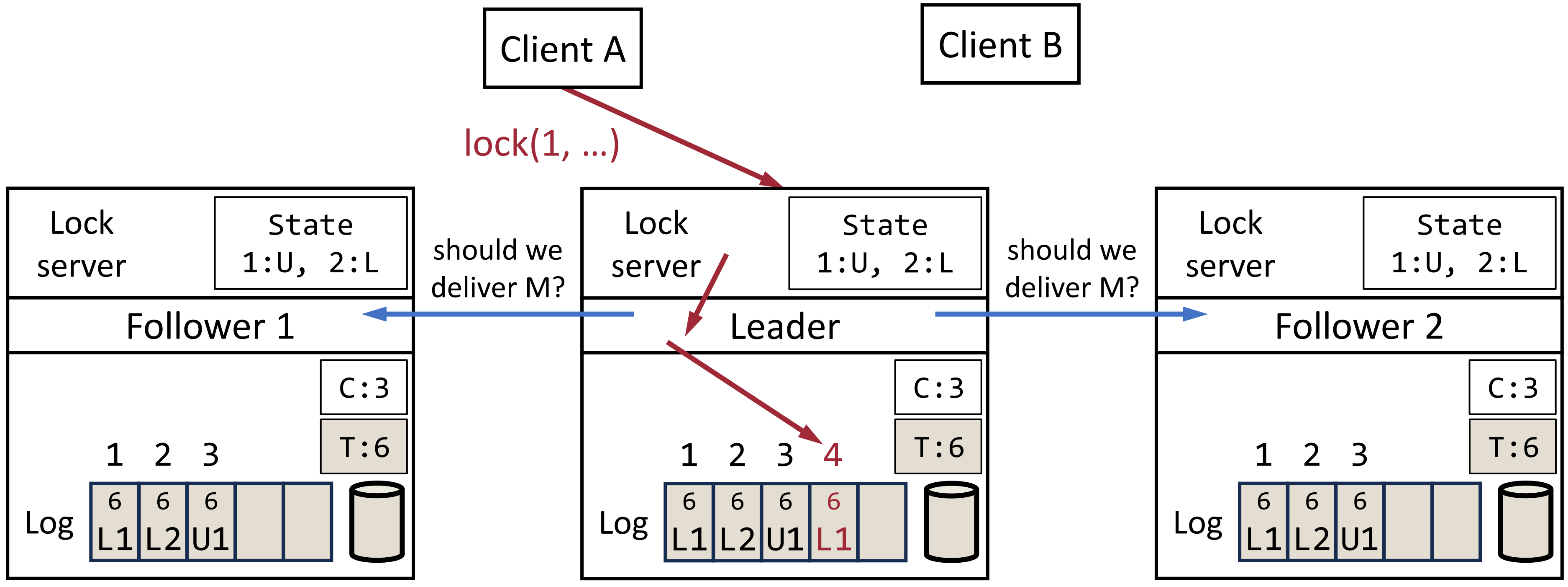
Replicated lock service

- Leader logs operation durably on disk



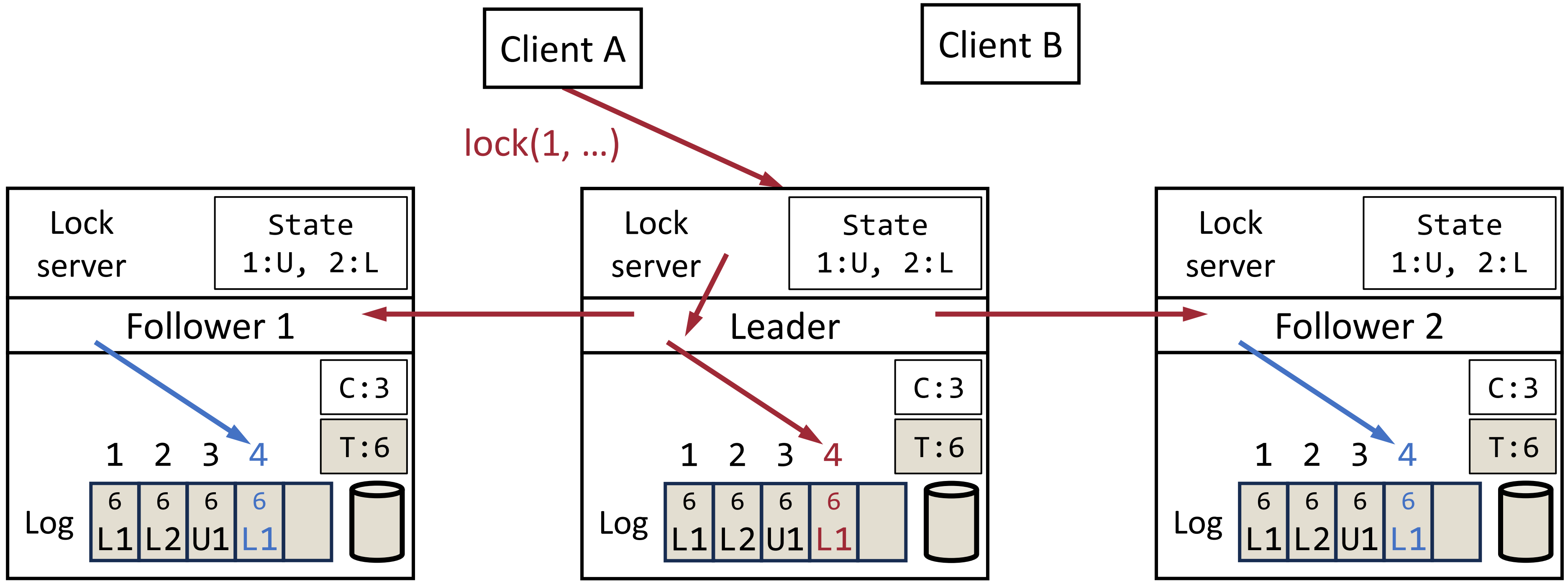
Replicated lock service

- Leader sends log replication RPC to followers
  - RPC is called AppendEntries (append operation entries to log)



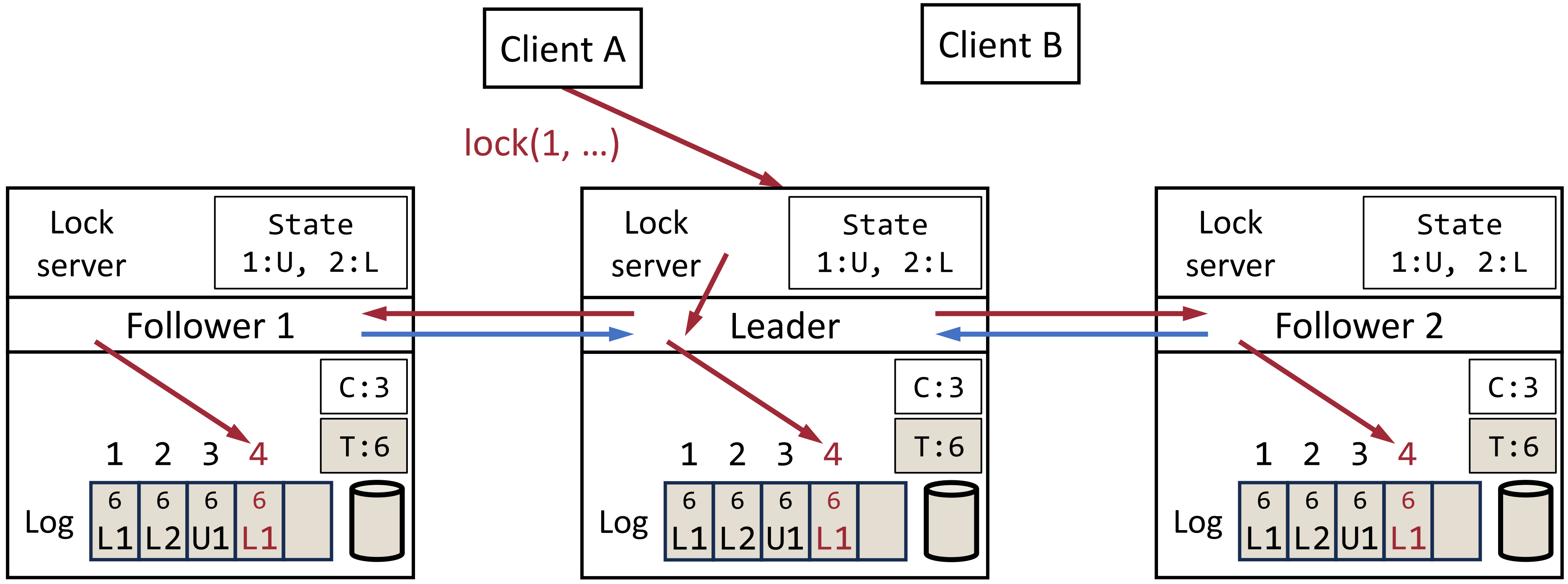
Replicated lock service

- Followers log operation durably on disk
- Operation from current term is committed when majority have logged it, will not be lost, even if all replicas fail



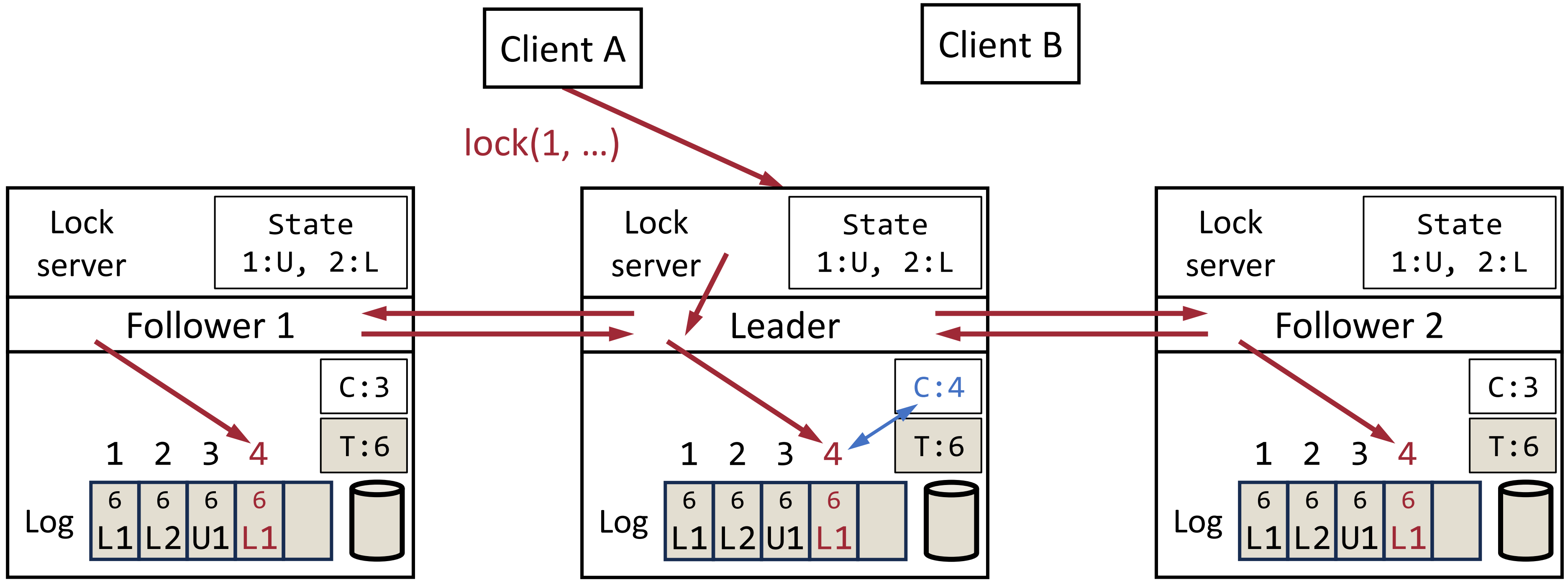
Replicated lock service

- Followers ack AppendEntries RPC to leader



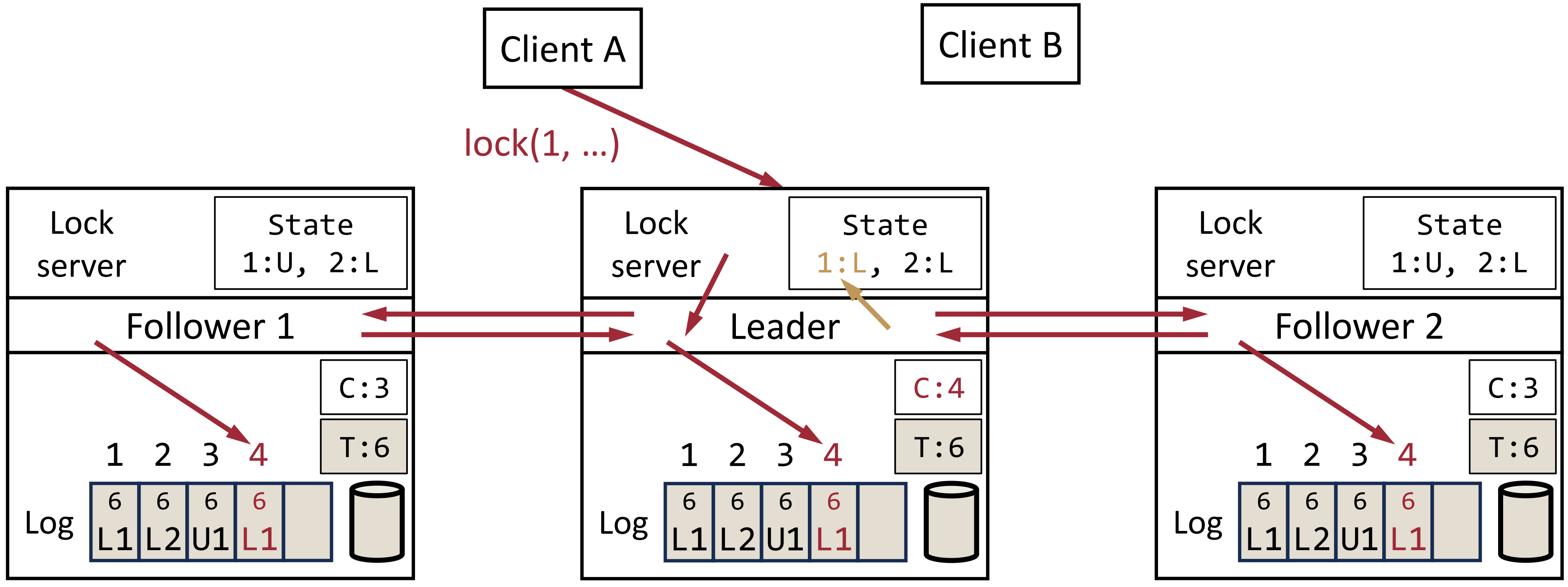
Replicated lock service

- Leader learns operation is committed when it receives AppendEntries acks from a majority (including itself)



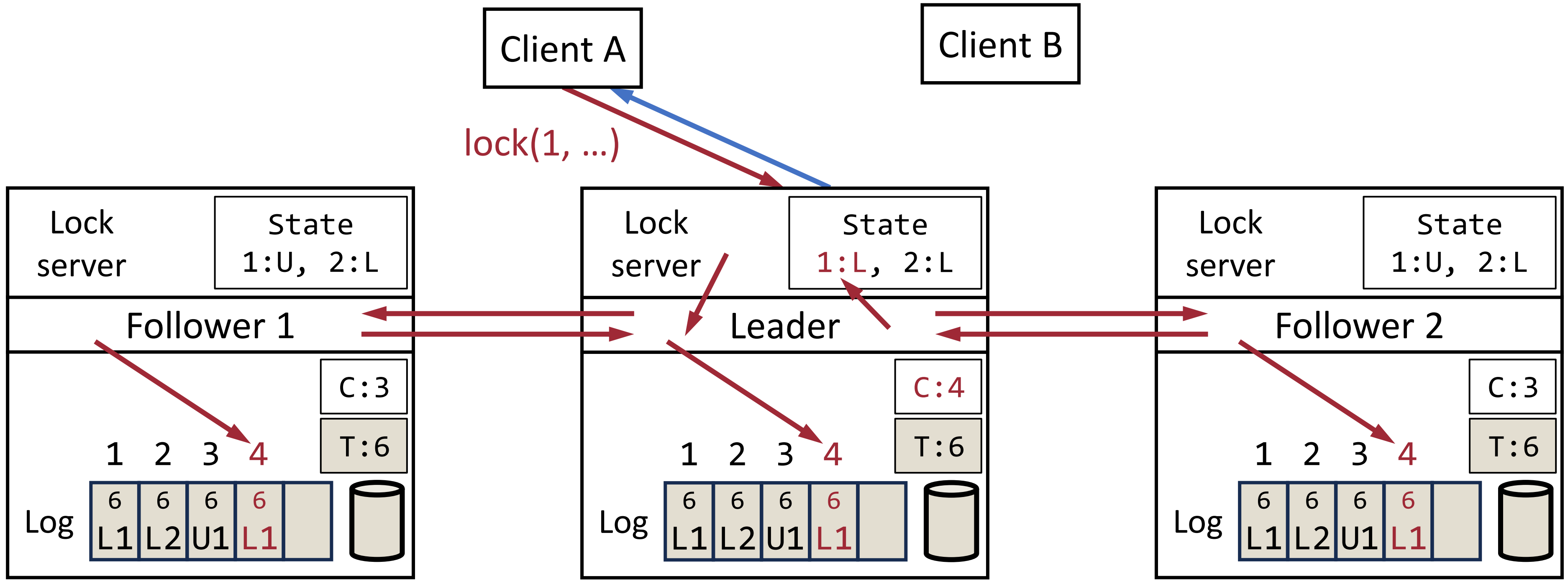
Replicated lock service

- Leader delivers committed operation to lock server
- Leader's lock server executes operation, updates its state



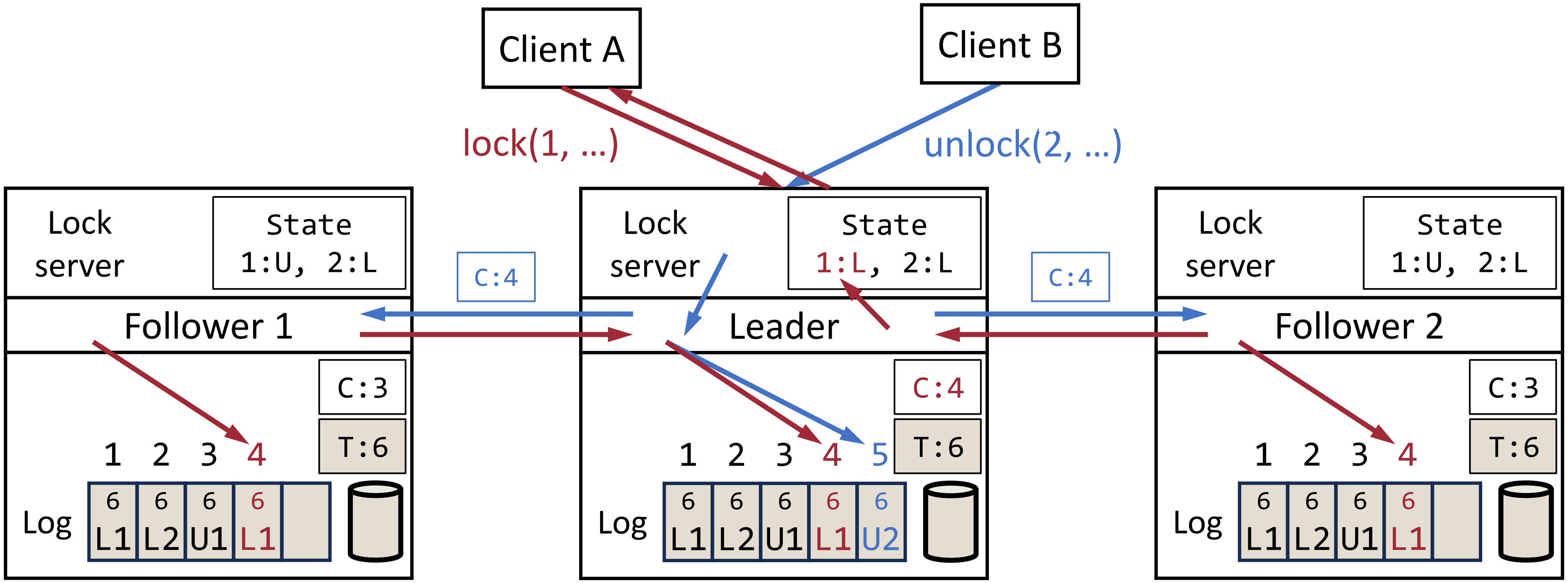
Replicated lock service

- Lock server acknowledges lock operation to client



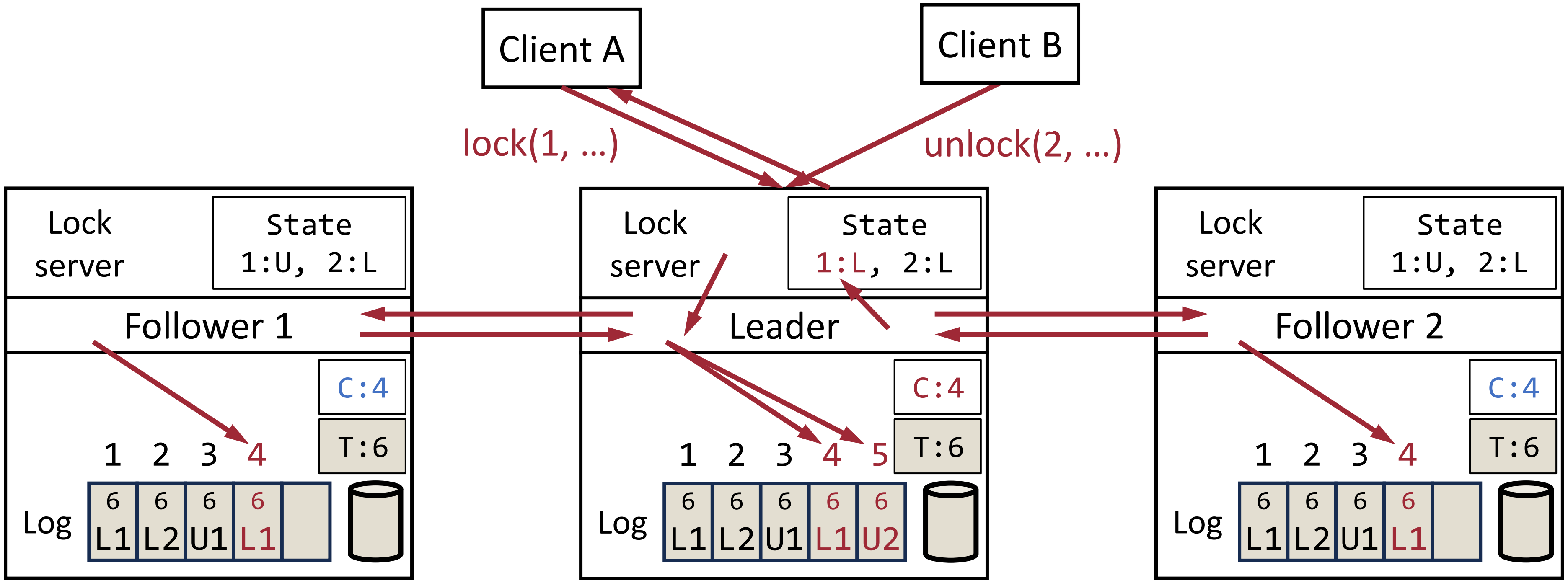
Replicated lock service

- Leader piggybacks commit info for operation when it sends AppendEntries RPC to followers for later operations



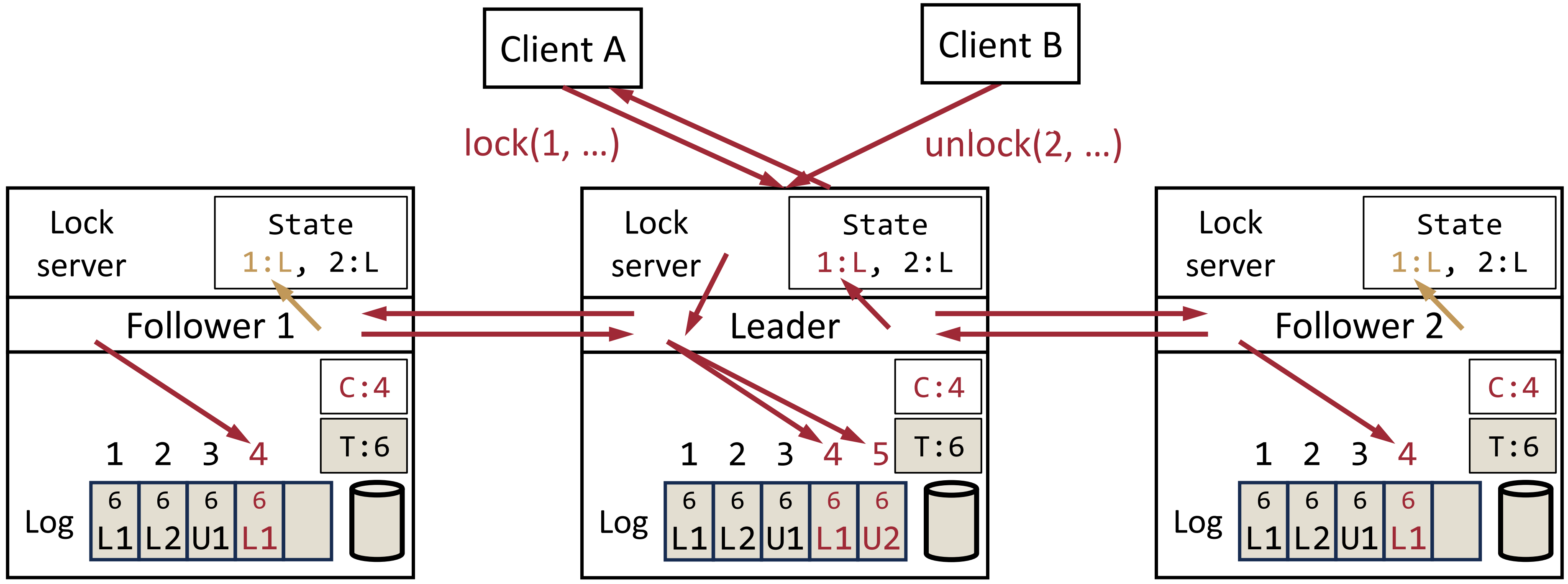
Replicated lock service

- Followers learn and update their commit info



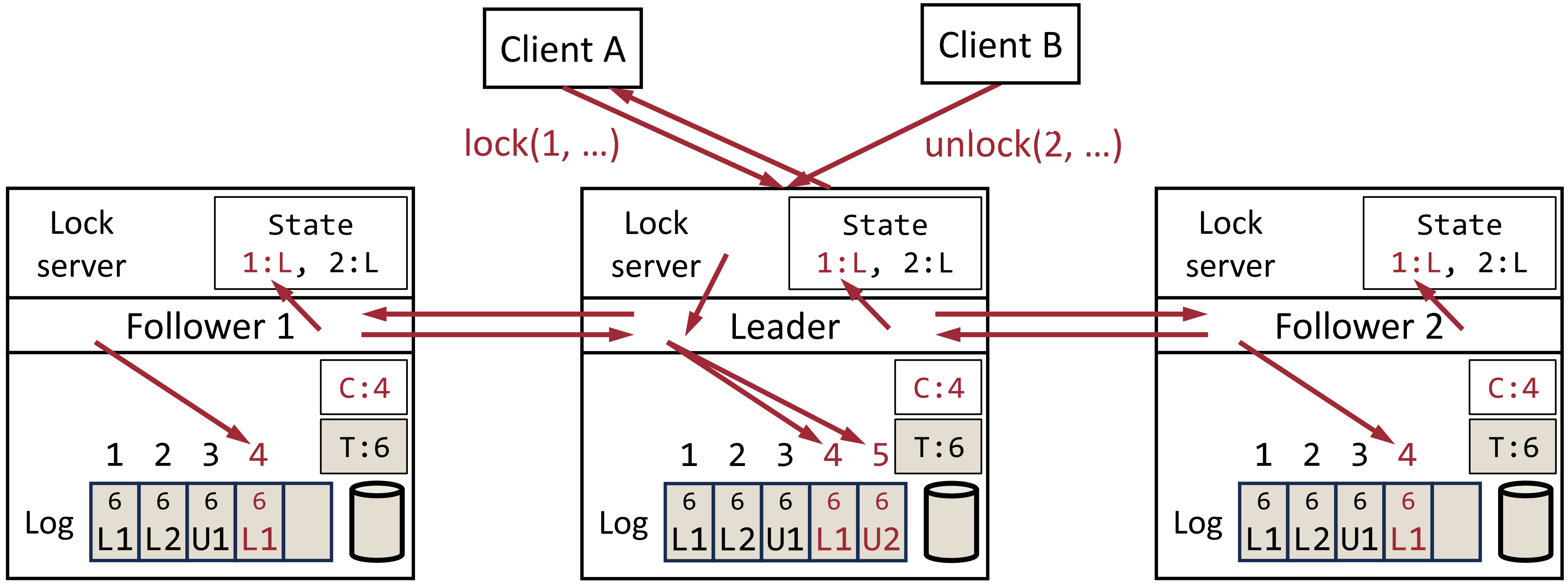
Replicated lock service

- Followers deliver committed operation to their lock server
- Follower's lock server executes operation, updates its state



Replicated lock service

- Now lock server state is consistent on the replicas

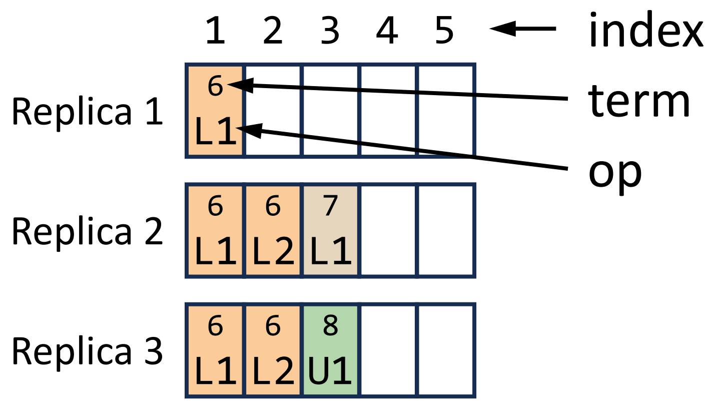


Why use logs?

- Lock service keeps current state of each lock
  - Why maintain a log (history of operations) as well?
- Log allows leader to order the operations
  - Follower logs may lag leader log, but eventually converge
- Log allows storing both tentative, committed operations
  - Tentative operations may commit or may be lost
  - Replicas only deliver committed operations to service
- Log allows handling failures
  - Leader can resend logged operations to unavailable followers
  - When replicas crash, they can recover their service state by replaying log from persistent storage on reboot

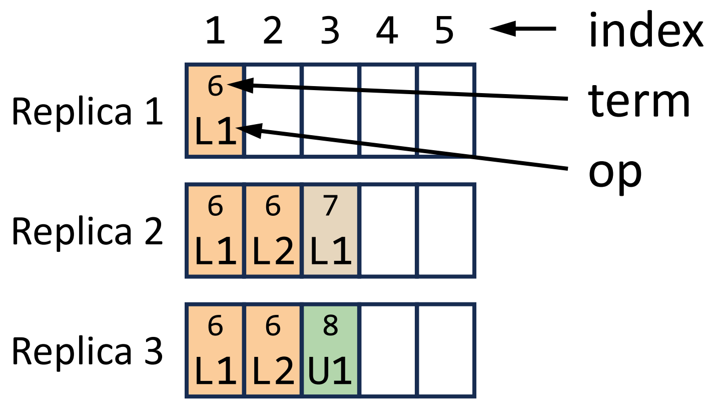
Log divergence

- After failures, logs at different replicas can diverge
- How can this happen?
  - R2 is leader in T6
    - Crashes before it can send L2 at I2 to R1
  - R2 reboots, becomes leader in T7
    - Logs L1 at I3, crashes
  - R3 becomes leader in T8
    - Logs U1 at I3
    - I3 entries in R2 and R3 conflict!



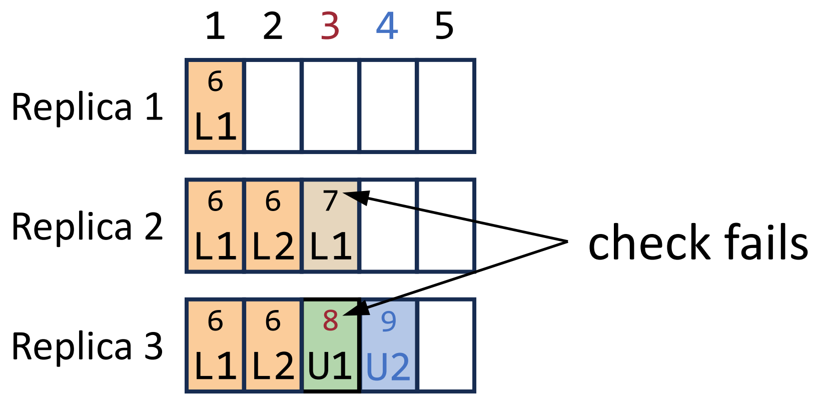
Log synchronization

- Raft forces followers to synchronize with leader's log
  - Ensures that a committed operation is at same index in all logs
- Raft always maintains these log matching properties
  - If two log entries on different replicas have same index and term:
    - They store the same operation
    - Logs are identical in all preceding entries
  - If an entry is committed, preceding entries are also committed



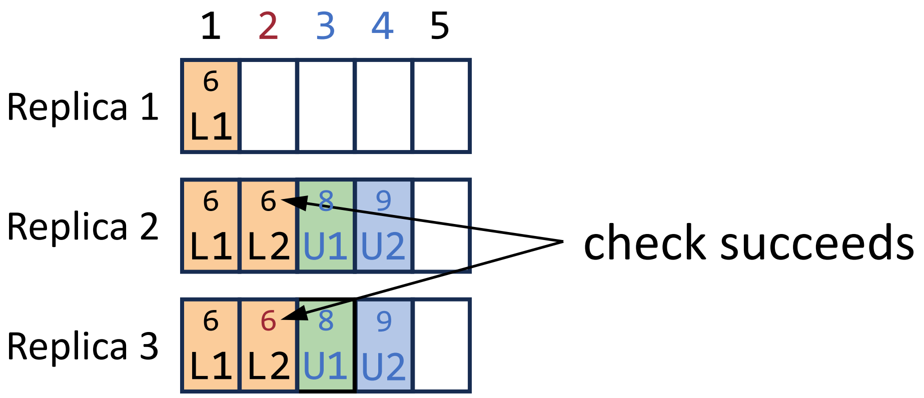
Log synchronization example

- Say R3 is leader in T9, logs U2 at I4
- R3 sends AppendEntries RPC to R1 and R2
  - Sends U2 at I4
  - Includes Term 8 of previous entry (Entry at I3)
- R2 checks match for previous entry
  - Term check fails (7 != 8)
  - R2 returns failure to R3



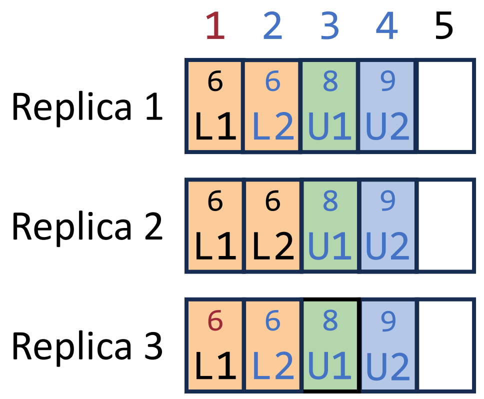
Log synchronization example

- R3 resends AppendEntries RPC to R2
  - Sends U1 at I3 and U2 at I4 to R2
  - Includes Term 6 of previous entry (Entry at I2)
- R2 checks match for previous entry
  - Term check succeeds (6 == 6)
  - R2 applies U1 at I3 and U2 at I4
  - All previous entries match due to log matching property



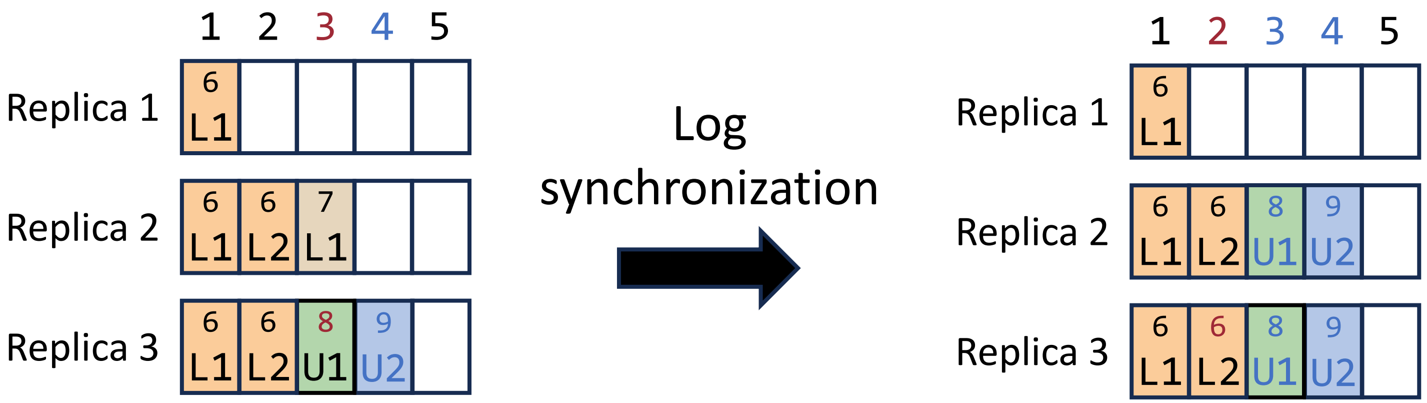
Log synchronization example

- Similarly, R3 resends AppendEntries RPC twice to R1 to send entries at I2, I3, I4
- Result: followers delete and synchronize the tail of their log that differs from the leader



Understanding log synchronization

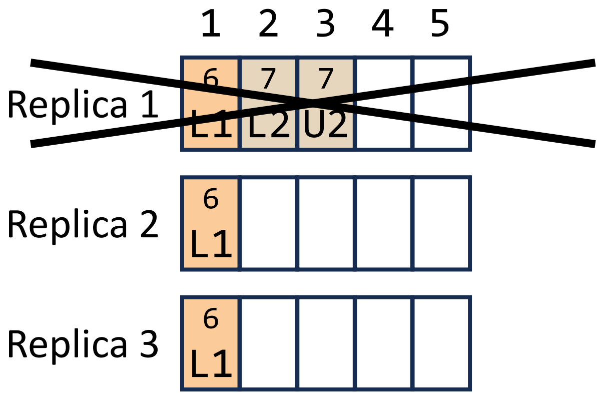
- Why is it okay for R2 to rollback its L1 at I3?



- What entries cannot be rolled back?
    - Committed entries, since client may have seen a reply for them
- Leader cannot forget a committed entry
    - Leader’s log must have all previously committed entries

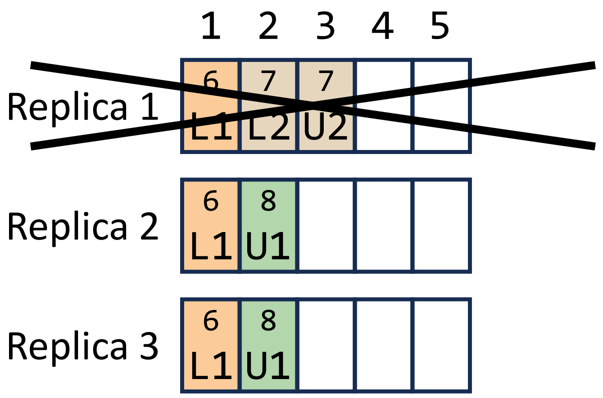
Is entry committed?

- R1 is leader in T7, logs entries at I2 and I3, then crashes



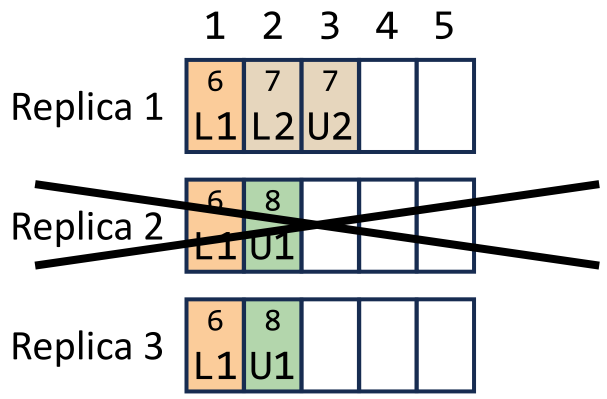
Is entry committed?

- R1 is leader in T7, logs entries at I2 and I3, then crashes
- R2 becomes leader in T8, replicates entry at I2 to R2 & R3



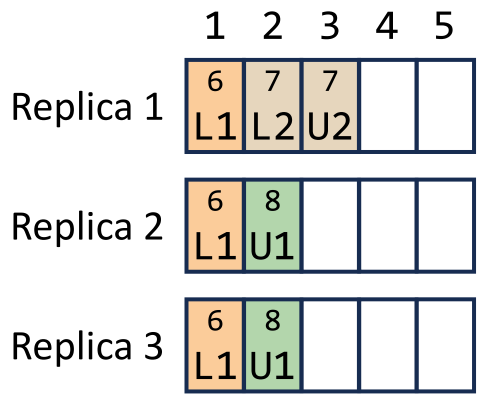
Is entry committed?

- R1 is leader in T7, logs entries at I2 and I3, then crashes
- R2 becomes leader in T8, replicates entry at I2 to R2 & R3
- Then, R2 crashes, R1 reboots
- Can R1 and R3 determine whether U1 at I2 is committed?
- Can R1, with the longest log, becomes the leader?



Restriction during leader election

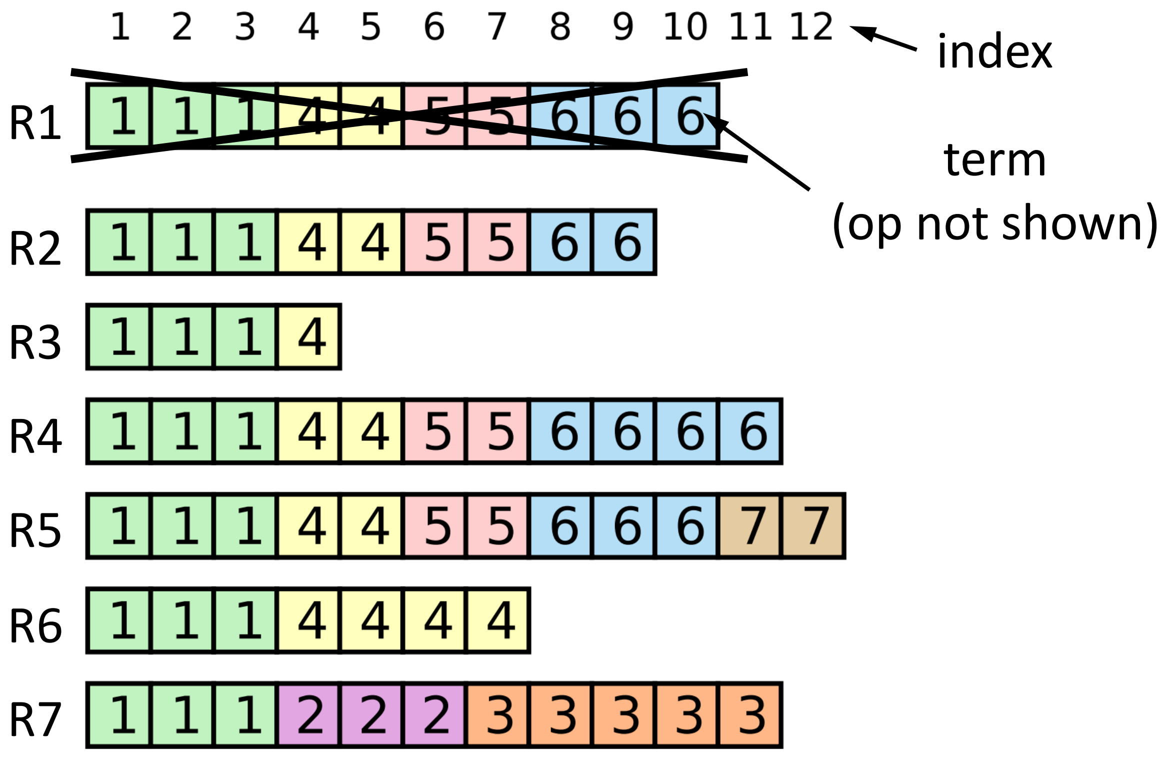
- Recall, candidate becomes leader when it receives votes from majority of replicas
- Raft adds a restriction so a candidate can only become a leader if it has all potentially committed entries
- Replicas respond to candidate that is at least as up to date:
  - Candidate has higher term in last log entry, or
  - Candidate has same last term and same or longer log length



With leader restriction,
only R2 and R3 can become leaders

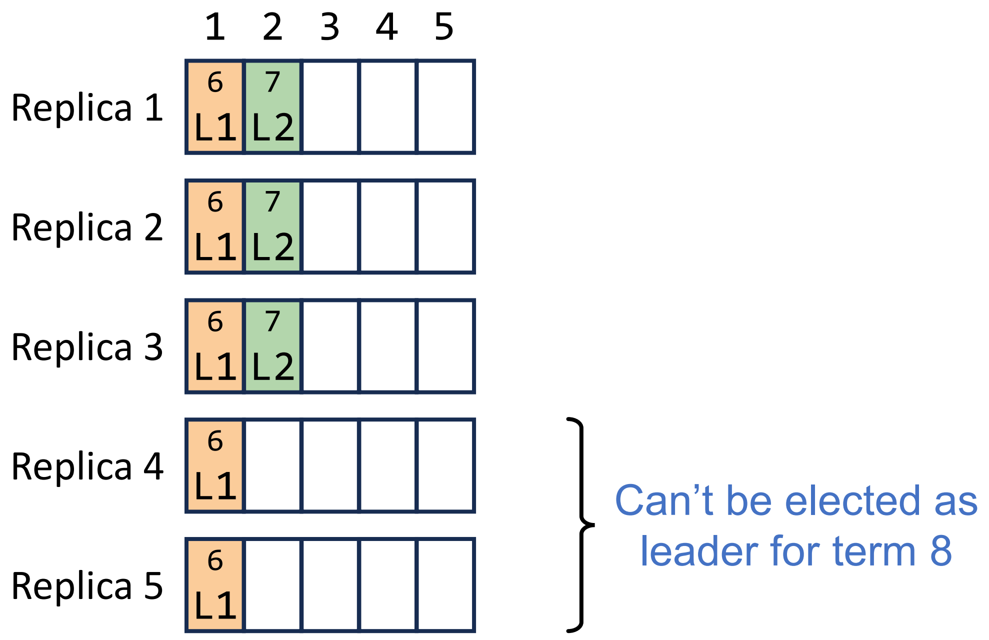
Leader restriction example

- Say leader R1 has crashed, which replicas can be leaders?
  - R2, R4 and R5 can get votes from at least 4 replicas
  - Is it okay if R2 becomes leader (though R4, R5 have longer logs)?



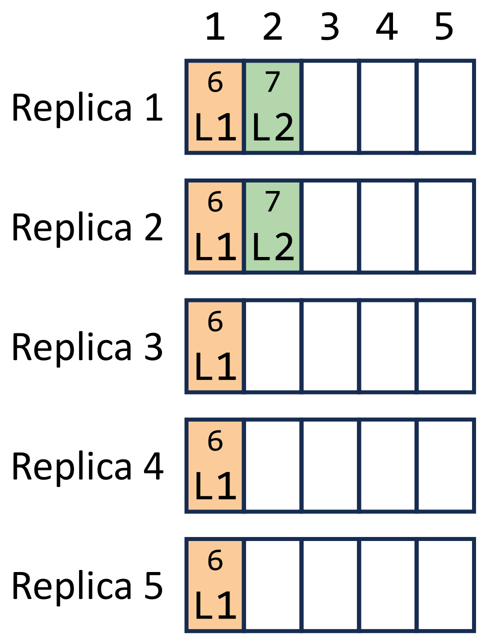
When does a leader commit an entry?

- Leader in Term 7 is Replica 1
- Leader knows any entry of current term (e.g., at I2) is committed when it is stored durably on a majority
- This is safe since leader in Term 8 must contain entry at I2



Committing entry of previous term

- Say Leader R1 replicates L2 at I2 to R1 and R2



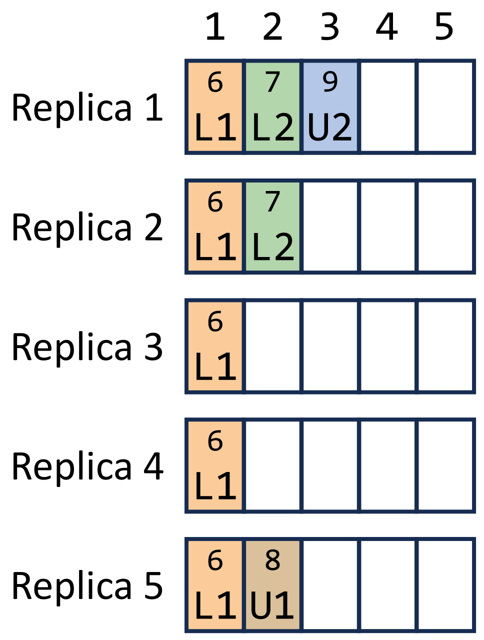
Committing entry of previous term

- Say Leader R1 replicates L2 at I2 to R1 and R2
- Then R5 becomes leader at T8, creates U1 at I2

	1	2	3	4	5
Replica 1	6 L1	7 L2			
Replica 2	6 L1	7 L2			
Replica 3	6 L1				
Replica 4	6 L1				
Replica 5	6 L1	8 U1			

Committing entry of previous term

- Say Leader R1 replicates L2 at I2 to R1 and R2
- Then R5 becomes leader at T8, creates U1 at I2
- Then R1 becomes leader at T9, creates U2 at I3



Committing entry of previous term

- Leader R1 at T9 replicates L2 at I2 to R3, then crashes
  - Entry 2 is now on a majority of servers, is it safely committed?
- R5 can be elected as leader for Term 10 (how?)
  - If elected, it will overwrite L2 at I2 on R1, R2, and R3!

	1	2	3	4	5
Replica 1	6 L1	7 L2	9 U2		
Replica 2	6 L1	7 L2			
Replica 3	6 L1	7 L2			
Replica 4	6 L1				
Replica 5	6 L1	8 U1			

Raft’s commit rule

- A leader decides that an entry in current or previous term is committed when:
  - Entry is stored on a majority
  - At least 1 new entry from leader’s term is also in majority

	1	2	3	4	5
Replica 1	6 L1	7 L2	9 U2		
Replica 2	6 L1	7 L2	9 U2		
Replica 3	6 L1	7 L2	9 U2		
Replica 4	6 L1				
Replica 5	6 L1	8 U1			

Crash Recovery

Handling crash failures

- When 1 in 3 replicas crash, Raft can continue operation
  - But crashed replica should be repaired soon
  - Otherwise, a second replica failure will lead to unavailability
- Two types of failures
  - Replica crashes permanently (crash-stop)
    - Use a new server as replica
    - Transfer entire log from leader to new server, may take a while
  - Replica crashes, reboots, disk data survives (crash-recovery)
    - Any or all replicas may crash due to power failure
    - Transfer recent log entries from leader to replica

Durable state in Raft

- Each replica stores following state on persistent storage (e.g., disk, SSD, etc.)
  - Log: stores committed (and tentative) entries
    - If committed entries are lost from a majority of replicas, then they could be forgotten by a leader in a later term
  - votedFor: stores candidate that replica voted for in current term
    - If lost after reboot, then replica could vote for another candidate in the same term, could lead to more than one leader in same term
  - Current term: stores latest term known to replica
    - Needed for votedFor
    - Avoids voting for or responding to a superseded leader

How to access durable state?

- State on disk is cached in memory
  - If state is cached at startup, it does not need to be read again
- When should state be stored to disk?
  - After it is modified
  - Before sending RPC or RPC response
  - Why?
- Storing state durably is expensive
  - 10 ms on disk, 0.1 ms on SSD, limits throughput to 10^2-10^4 ops/s
- Various optimizations possible
  - Use battery-backed RAM or persistent memory
  - Batch multiple log entries per disk write

Simple crash recovery

- After a replica crashes and reboots, in-memory state of (e.g., lock) service needs to be reinitialized
    - Raft replays log on disk to create service state
- Each Raft replica stores volatile state
    - commitIndex: highest log entry known to be committed
    - lastApplied: last log entry applied to state machine
- Leader stores volatile state
    - nextIndex[r]: index of next entry to send to Server r
    - matchIndex[r]: index of highest entry known to be replicated at Server r
- After reboot, a replica initializes its volatile state so that log replication replays entire log to recreate service state
    - E.g., lastApplied, commitIndex, matchIndex start at 0 on reboot

Log Compaction

Growing log size

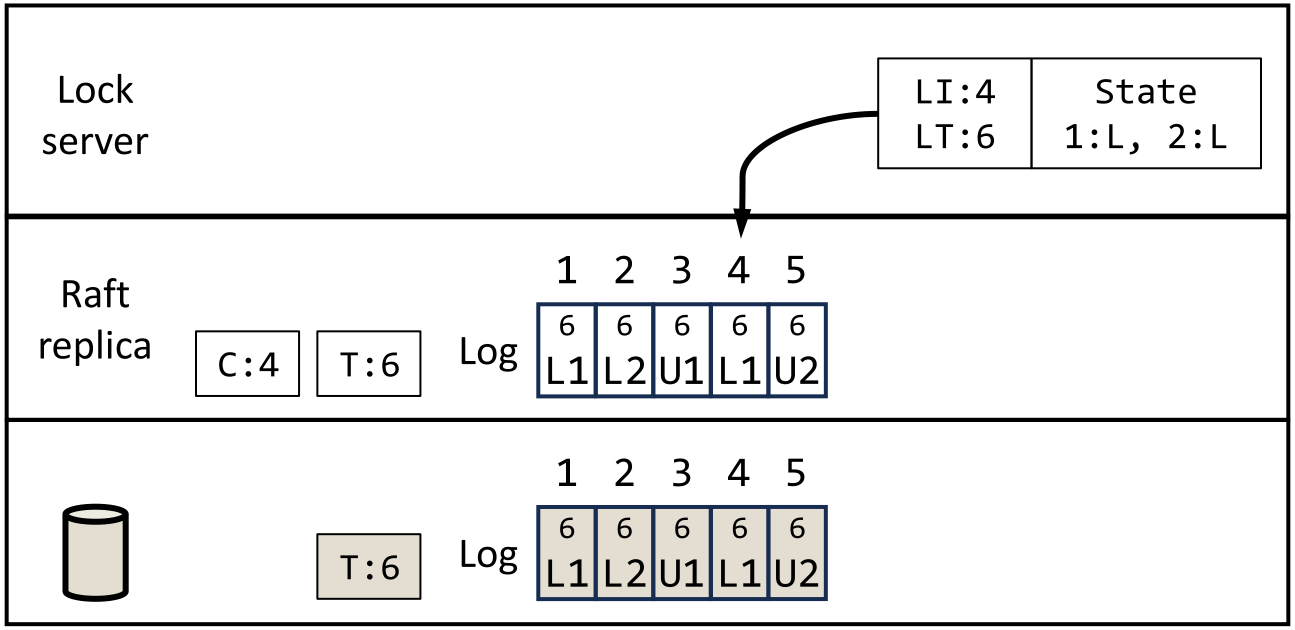
- Log size will grow over time
  - Occupies disk (needs more space)
  - Crash recovery replays entire log (takes more time)
  - Leader sends entire log to new server (takes more time)
- Log size can be much larger than service state
  - But clients only see service state, not log

Reducing log size

- How can we reduce the log size?
- Intuition:
  - Persist a snapshot of the service state to disk
  - Keep only the tail of the log after the snapshot

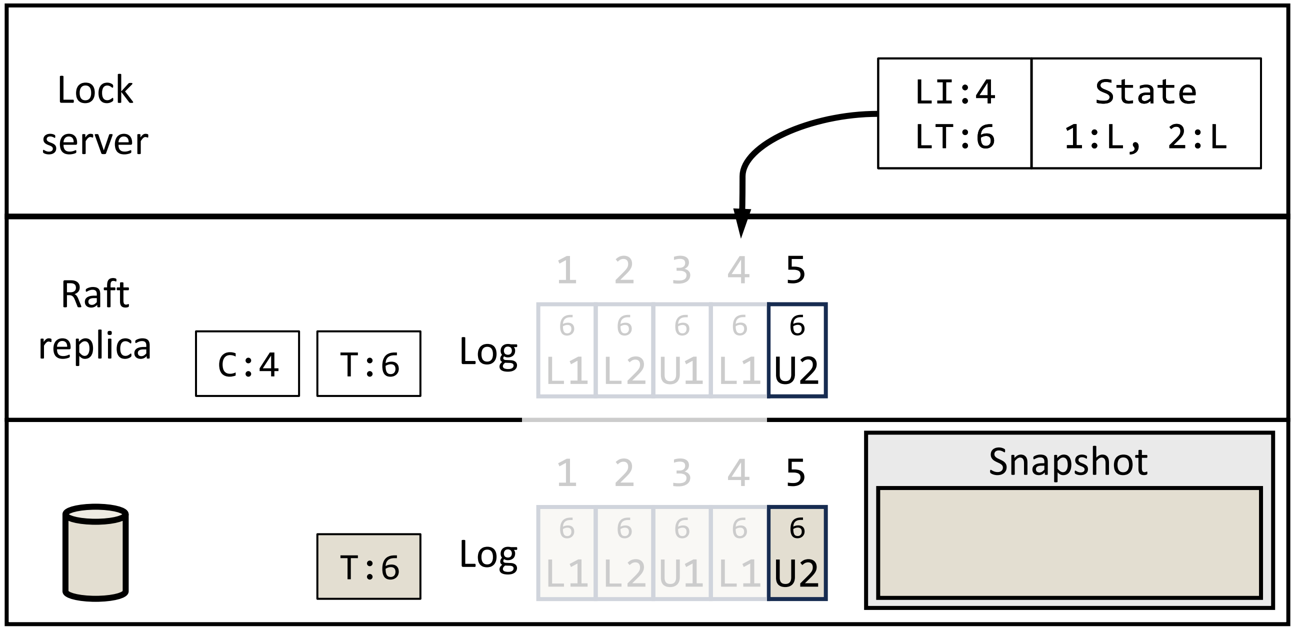
Snapshots and log compaction

- Service provides to Raft
  - Snapshot of its state
  - Last <log index, term> included in the snapshot



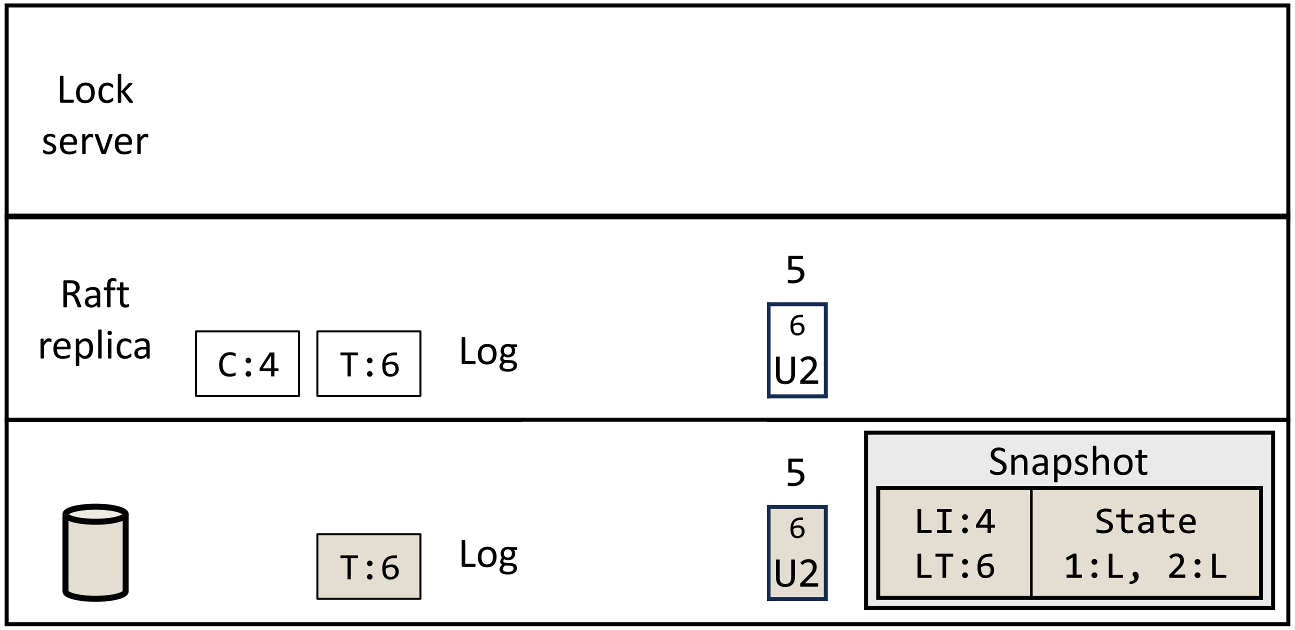
Snapshots and log compaction

- Raft persists snapshot state, last <log index, term>
- Then discards log until snapshot log index (log compaction)



Snapshots and recovery

- After crash, during recovery
  - Service loads snapshot state into memory
  - Raft uses last <log index, term> to send tail of log to service



Snapshot RPC

- Every replica has a log
- Every replica (not just leader) snapshots independently
- Problem: If leader compacts its log while follower is offline, follower’s log may end before the start of leader’s log
    - But leader only sends entries from its log to followers
- Solution: Leader sends its snapshot (InstallSnapshot RPC) to a slow follower, then can continue sending its log

Client interaction

Client operations

- Clients send operations to leader, if leader unknown, contact any server, server redirects clients to leader
- Problem:
    - Suppose leader executes client operation, then crashes before sending response to client
    - Client retries same operation with another leader
    - Operation is executed twice
    - For linearizability, we need an operation to execute exactly once

Ensuring exactly-once semantics

- Client embeds unique request ID in each operation
- State machine performs duplicate detection
  - Keeps [client -> (request ID, response)] map
    for latest operation executed for the client
  - When Raft delivers an operation to the state machine,
    state machine checks if it has seen the client's request ID,
    and returns response (without re-executing operation)

Read-only operations

- Can a read-only operation be issued to any follower?
  - A follower can lag a leader, so the read may not read the latest data (needed for linearizability)
- Can a leader respond to a read-only operation without contacting any followers?
  - A leader doesn’t know whether it has been superseded
- In Raft, when leader receives a read-only operation:
  - Leader sends heartbeat messages to followers
  - Waits for a majority to know if it is still the current leader
  - Responds to read-only operation (no logging needed)
- An alternative is to use leases, see paper

Conclusions

- Raft uses a leader-based consensus scheme to implement fault-tolerant state machine replication
  - Ensures correctness by using majority when
    - Electing a leader (leader election)
    - Leader delivers messages (log replication)
  - Ensures liveness with randomized timers when electing leader
  - Provides linearizability consistency guarantees
  - Safety properties formally specified and proven
- A practical, heavily used implementation
  - Handles leader/follower crash-stop/crash-recovery failures
  - Log compaction, snapshots
  - Membership changes - adding/removing replicas, see paper

Wrap up

- This has been a long tour, but we finally have answers
  - Broadcast slides: Algorithms for all models, except total order broadcast, handle node failures. Later, we will look at fault-tolerant total order broadcast.
  - Linearizability slides: A single server can crash. Later, we will look at how to build a fault-tolerant replicated service that can ensure linearizability.
  - Replication slides: Fault tolerance in state-machine replication depends on the underlying total order broadcast protocol. Later, we will look at fault-tolerant total order broadcast.
- Raft is a fault-tolerant, total order broadcast protocol
  - Implements state machine replication, provides fault tolerance, ensures linearizability