

Consensus with RAFT

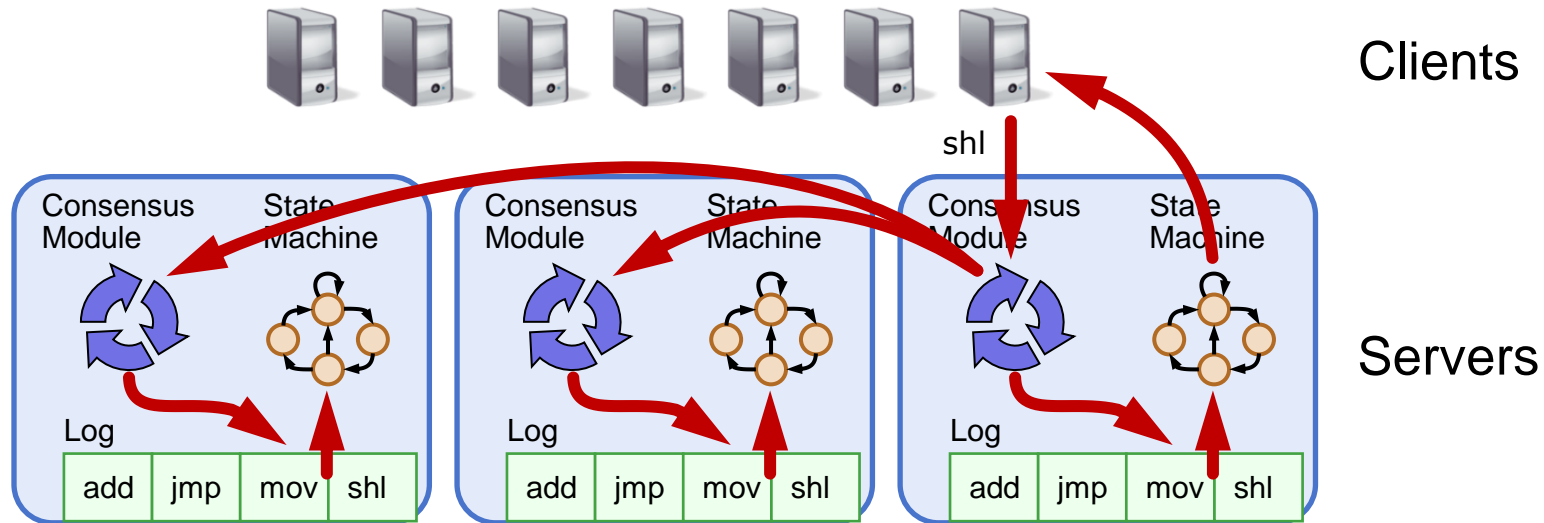
Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

These slides are modified versions of slides from Diego Ongaro, John Ousterhout
and Michael Freedman

Goal: Replicated Log



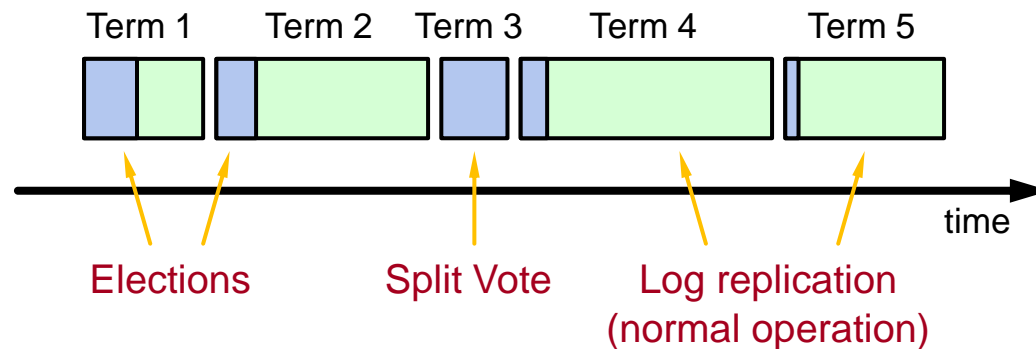
- Replicated log enables replicated state machine
 - All servers execute same commands in same order
- Consensus module ensures proper log replication

Raft Overview

- Raft is a library that uses a leader-based consensus scheme to implement fault-tolerant state machine replication
- Leader Election: ensures one leader at any time
- Log Replication: leader broadcasts messages to replicas in order (normal operation)
- Choosing Leader: ensures safety and consistency
- Client Interaction: ensures exactly-once semantics

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
 - Key role of terms: identify obsolete information

Server States

- At any given time, each server is either:
 - Leader: handles all client interactions, log replication
 - Follower: completely passive
 - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

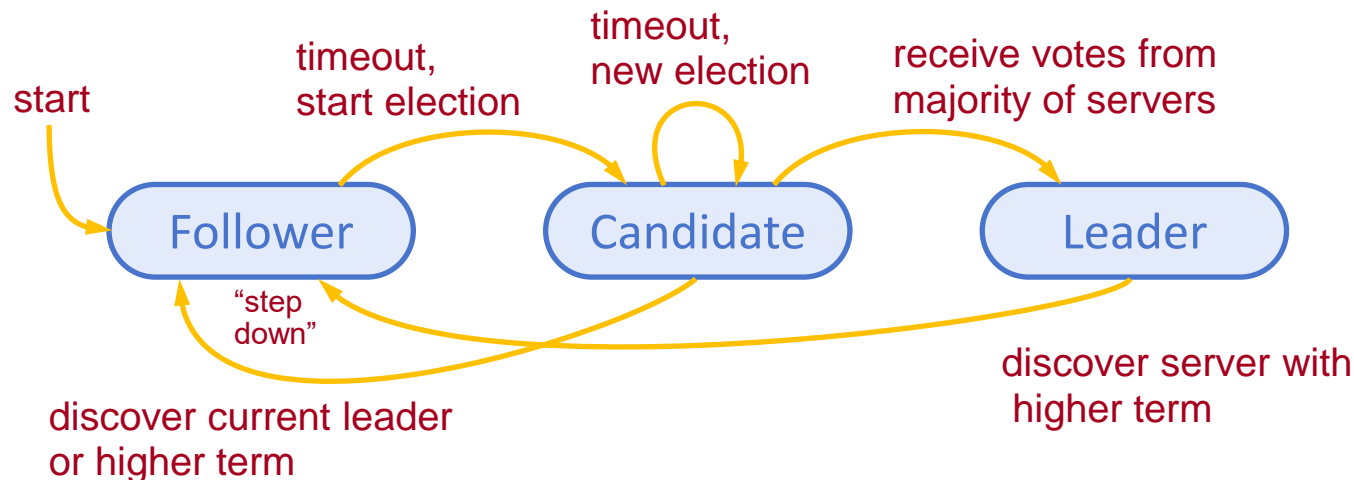
Follower

Candidate

Leader

Liveness Validation

- Servers start as followers
- Leaders send heartbeats (empty AppendEntries RPCs) to maintain authority over followers
- If electionTimeout elapses with no RPCs (100-500ms), follower assumes leader has crashed and starts new election

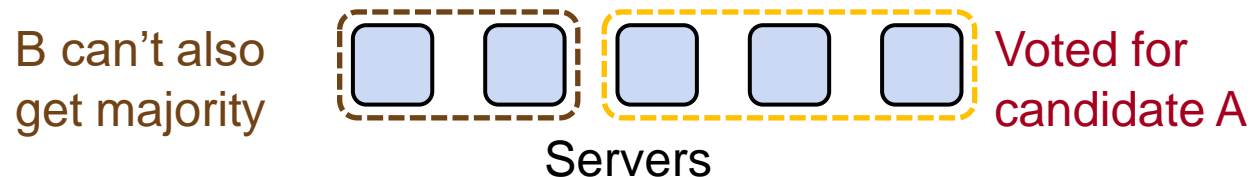


Elections

- Start election:
 - Increment current term, change to candidate state, vote for self
- Send RequestVote to all other servers, retry until either:
 - Receive votes from majority of servers:
 - Become leader
 - Send AppendEntries heartbeats to all other servers
 - Receive RPC from valid leader (with same or higher term):
 - Return to follower state
 - No-one wins election (election timeout elapses):
 - Increment term, start new election

Safety & Liveness

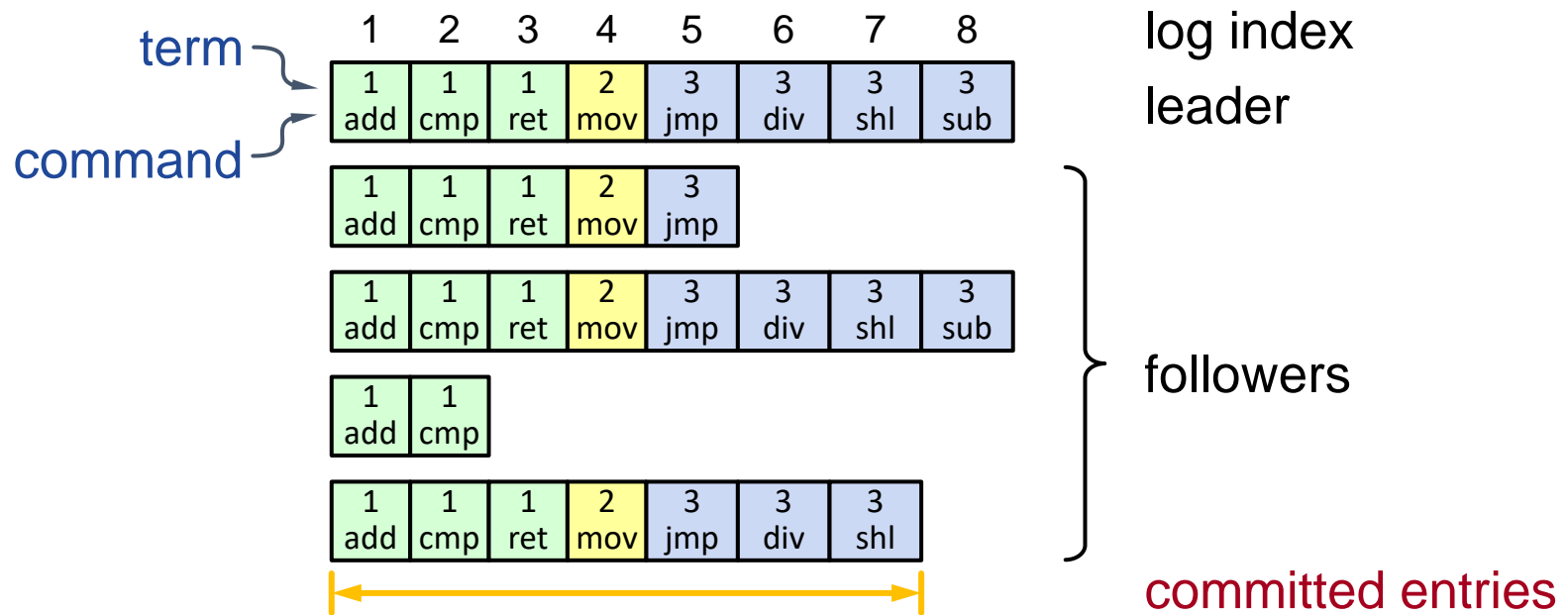
- Safety: allow at most one winner per term
 - Each server votes only **once** per term (**persists on disk**)
 - Two different candidates can't get majorities in same term



- Liveness: some candidate eventually wins
 - Each candidate chooses election timeouts randomly in $[T, 2T]$
 - One usually initiates and wins election before others start
 - Works well if $T \gg \text{network RTT}$

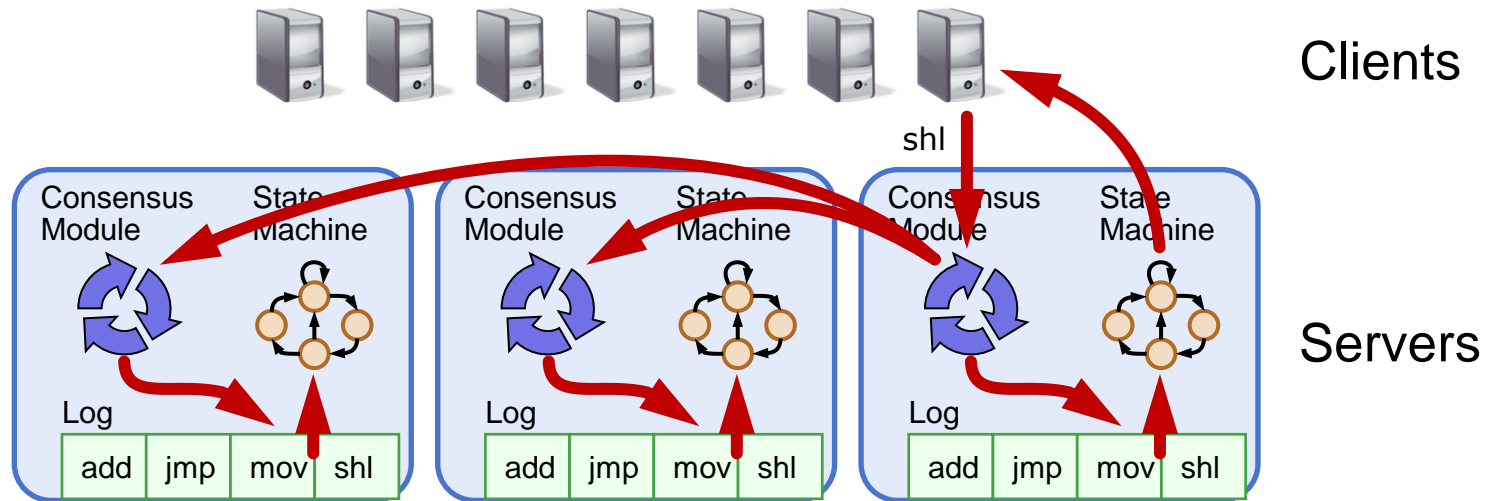
Log Replication

Log Structure



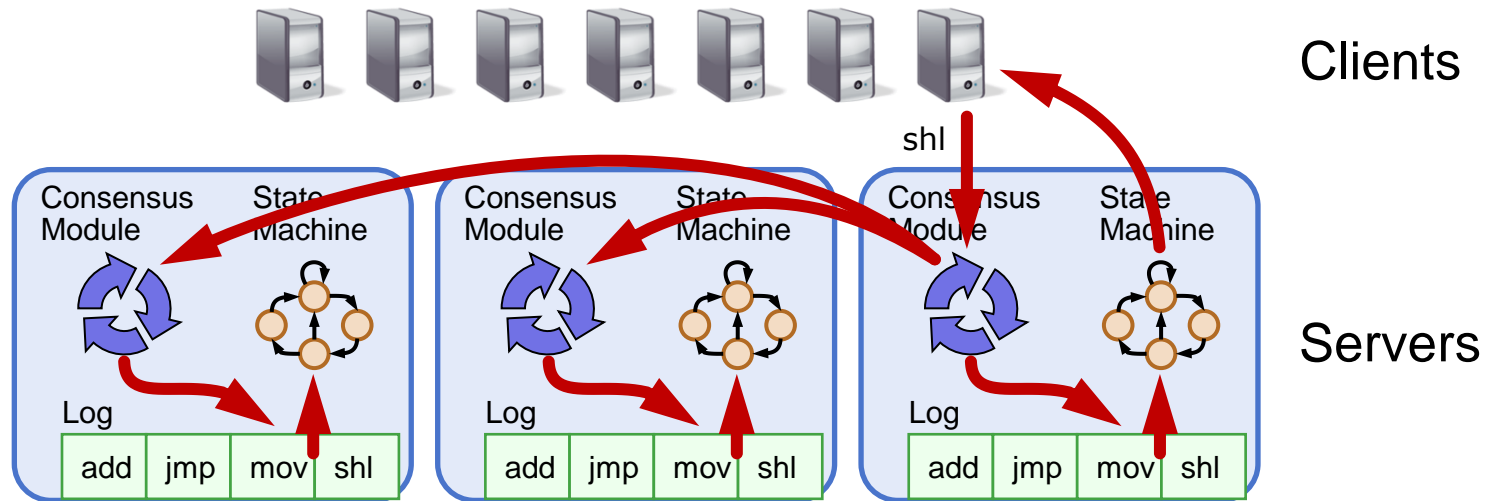
- Log entry = $\langle \text{index, term, command} \rangle$
- Log stored on stable storage (disk); survives crashes
- Entry created in **current term** is **committed** when it is **stored on majority of servers**
- Committed entry stored durably, eventually executed by state machines

Normal Operation



- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
 - Leader passes command to its state machine, sends result to client
 - Leader piggybacks commitment to followers in later AppendEntries
 - Followers pass committed commands to their state machines

Normal Operation



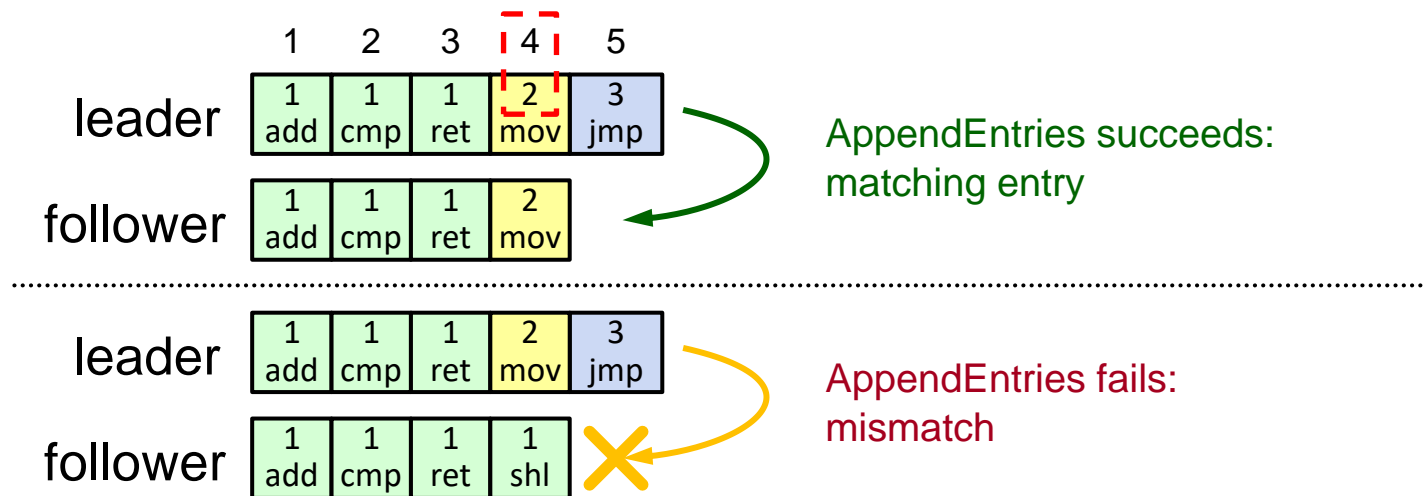
- Crashed / slow followers?
 - Leader retries RPCs until they succeed
- Performance is “optimal” in common case:
 - One successful RPC to any majority of servers

Log Operation: Highly Coherent

	1	2	3	4	5	6
server1	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
server2	1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

- If log entries on different servers have same index and term:
 - They store the same command
 - Logs are identical in all preceding entries
- If given entry is committed, all preceding entries are also committed

Log Operation: Consistency Check

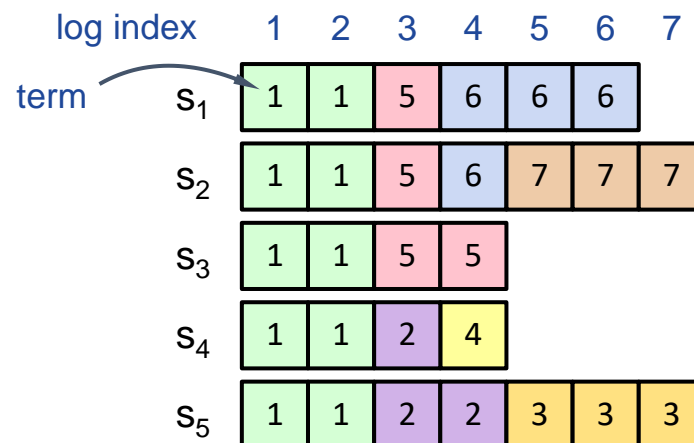


- AppendEntries has $\langle \text{index}, \text{term} \rangle$ of entry preceding new ones
- Follower must contain matching entry, or else it rejects
- Implements an induction step, ensures coherency

Choosing Leader

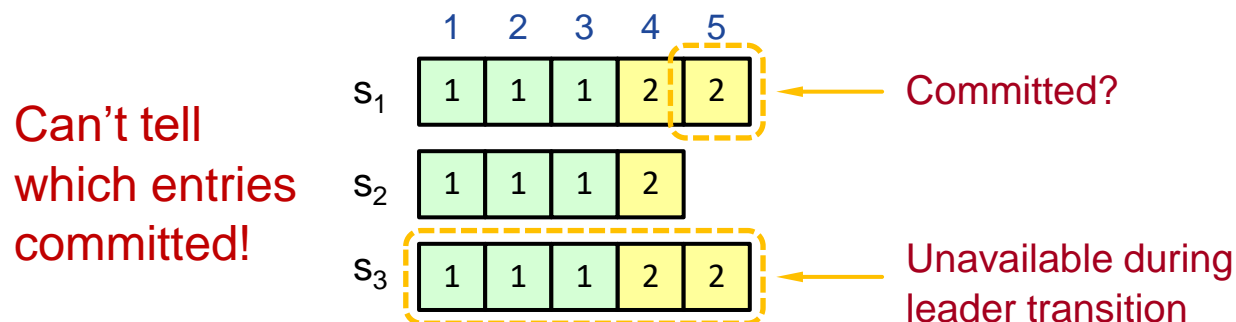
After Leader Change

- New leader's log is truth, no additional steps needed
 - Starts normal operation
 - Will eventually make follower's logs identical to leader's
 - Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous log entries



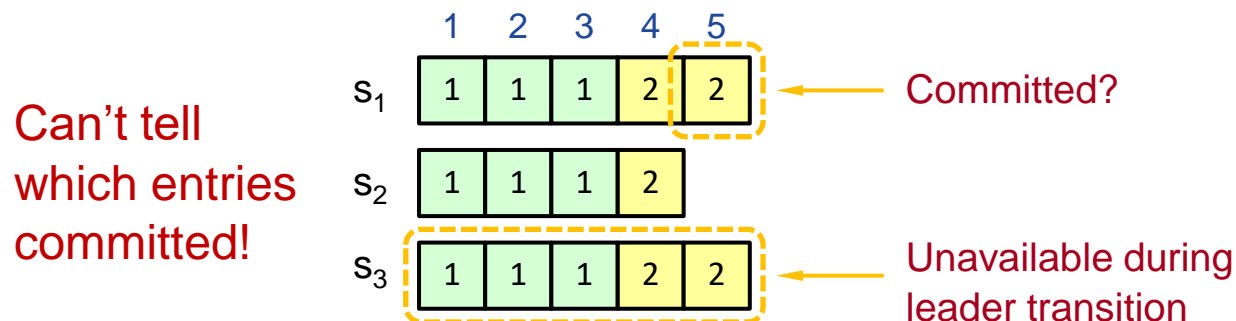
Choosing the Leader

- Raft leader completeness property: if log entry is **committed** in a term, entry will be present in logs of future term leaders
- Problem: how to determine which entries are committed?



Choosing the Best Leader

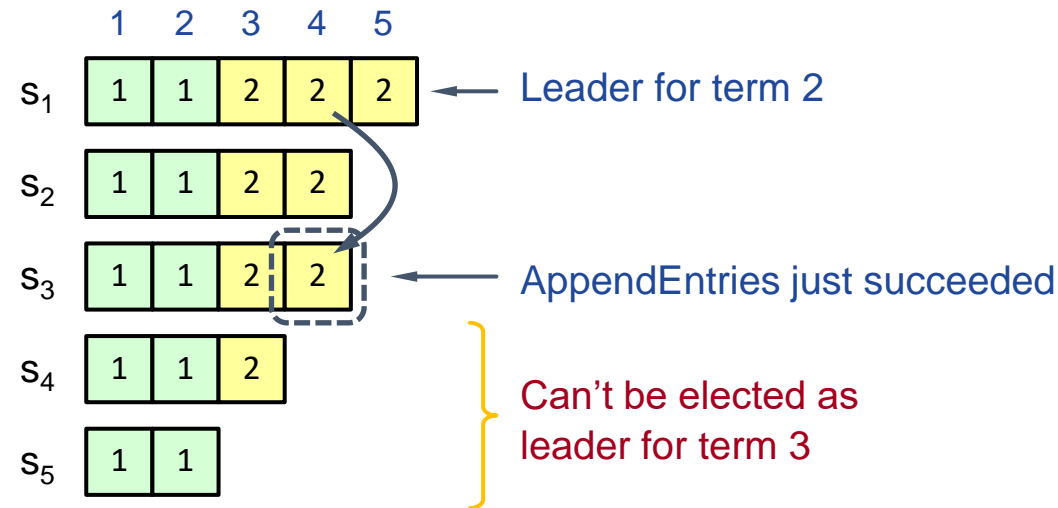
- Elect candidate that contains all **potentially** committed entries
 - In RequestVote, candidates incl. index + term of last log entry
 - Voter V denies vote if its log is “more complete”:
last entry has (higher term) or (higher index with same term)
 - Leader will have “most complete” log among electing majority



Leader's Commitment Rule

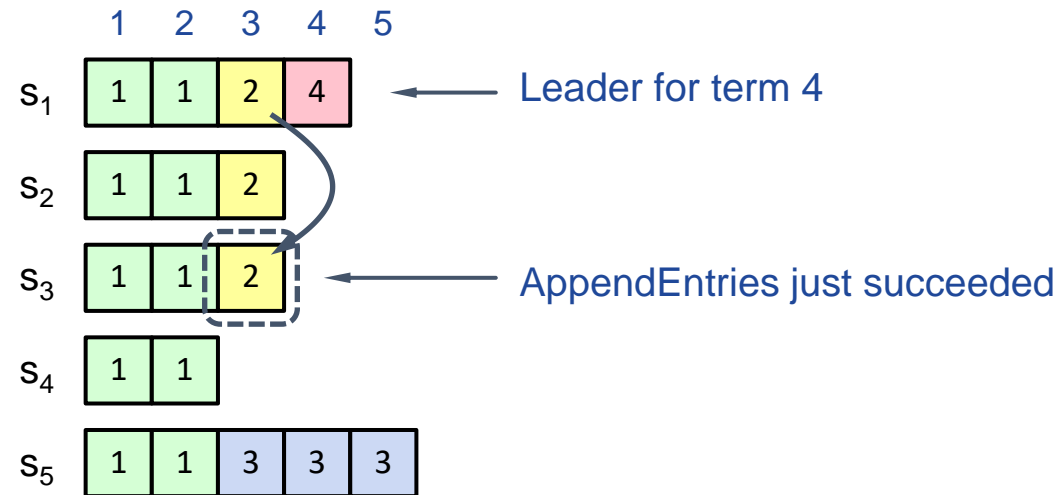
- Two cases:
 - Entries in current term
 - Entries in previous terms

Leader Commits Current Term Entry



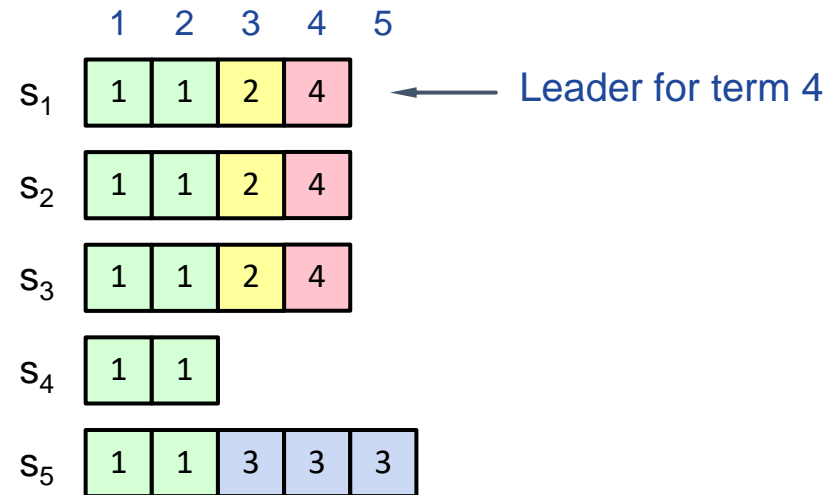
- Leader knows entry in current term is committed when it is stored durably on a majority
- This is safe because leader for Term 3 must contain Entry 4

Leader Commits Earlier Term Entry



- In Term 2, Entry 3 replicated to S₁ and S₂
- Leader 4 finishes replicating Entry 3 (from Term 2) to S₃
 - Entry 3 is now on a majority of servers
- Is Entry 3 safely committed?
 - S₅ can be elected as leader for Term 5 (how?)
 - If elected, it will overwrite Entry 3 on S₁, S₂, and S₃

New Commit Rules



- For leader to decide that an entry (in current or previous term) is committed:
 - Entry stored on a majority
 - At least 1 new entry from leader's term is also in majority
- E.g., once Entry 4 is committed, S₅ cannot be elected leader for Term 5, and Entry 3 and 4 are both safe

Client Interaction

Client Protocol

- Clients send commands to leader, if leader unknown, contact any server, which redirects client to leader
- Leader only responds after command logged (locally), committed, and executed by leader
- Problem: A leader could execute command and then fail before returning response to client
 - Client retries the same command with another leader, so command executed twice
- To ensure **exactly-once semantics**, state machines must perform duplicate detection
 - Client embeds unique request ID in each command
 - State machine checks request ID and returns previous result

Discussion

Q1

- A lease serves as a **lock with a timeout**
- Say, a leader is elected using leases as follows:
 - An external server stores
 - Server location of the current leader
 - Leader's lease, i.e, a time until which this server will serve as leader
 - Leader sends periodic heartbeats to external server
 - Each heartbeat renews the leader's lease, i.e., extends the time for which server will remain the leader
 - Other servers contact the external server to find leader
 - If the lease has expired, the first server to contact the external server becomes the leader
- What safety issues can occur with this leader election mechanism?

Q2

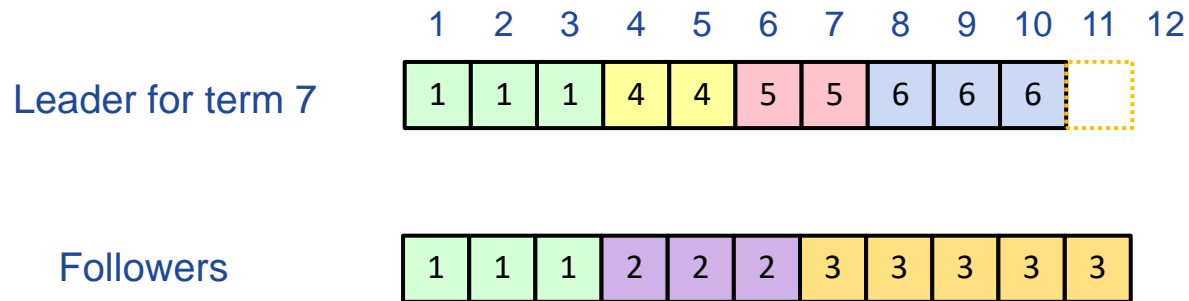
- Raft also uses heartbeats. Why doesn't it have these safety issues? Why can heartbeats cause liveness and/or availability issues?

Q3

- Besides the log, each server maintains the following persistently (on disk):
 - `currentTerm` (latest term that the server has seen)
 - `votedFor` (candidate that received vote in current term)
- Why are these values maintained on disk?

Q4

- When a new leader is elected, it may delete extraneous entries in a follower:



- Intuitively, why doesn't this cause data loss?

Q5

- What is exactly-once semantics and why is it useful?
Why is it hard to enforce?