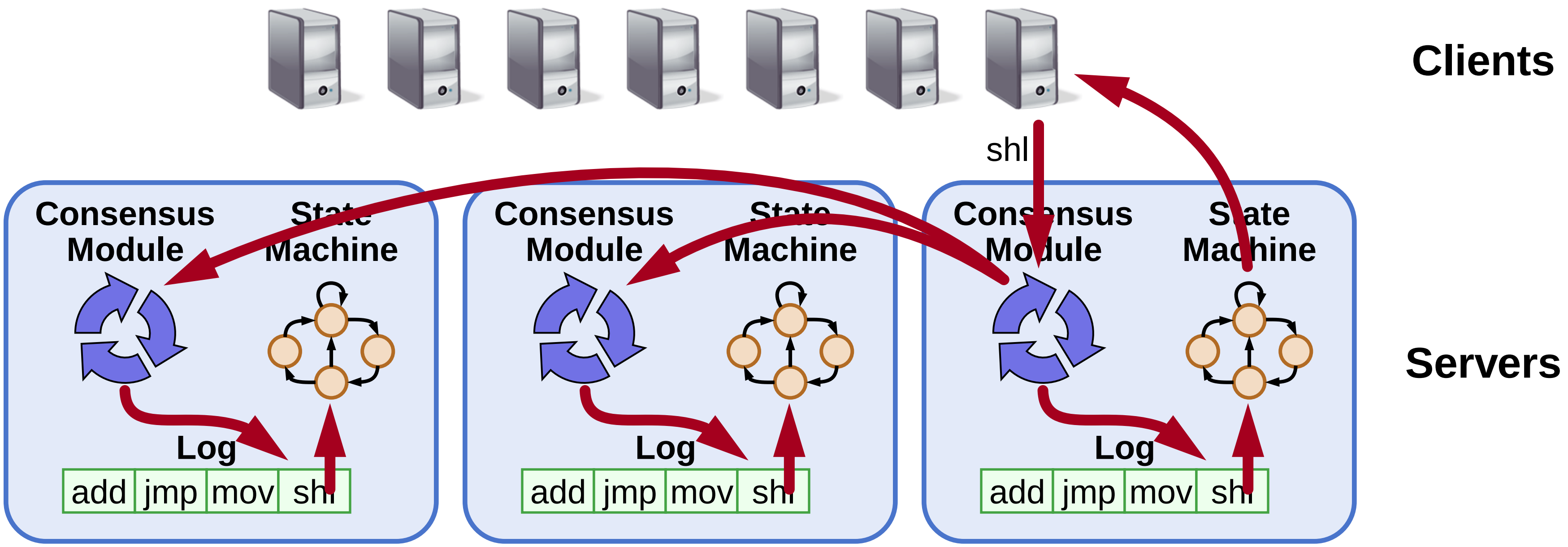
Raft: A Consensus Algorithm for Replicated Logs

Credits to Diego Ongaro and John Ousterhout

Stanford University

But with some of my modifications.

Goal: Replicated Log



- **Replicated log => replicated state machine**
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

Approaches to Consensus

Two general approaches to consensus:

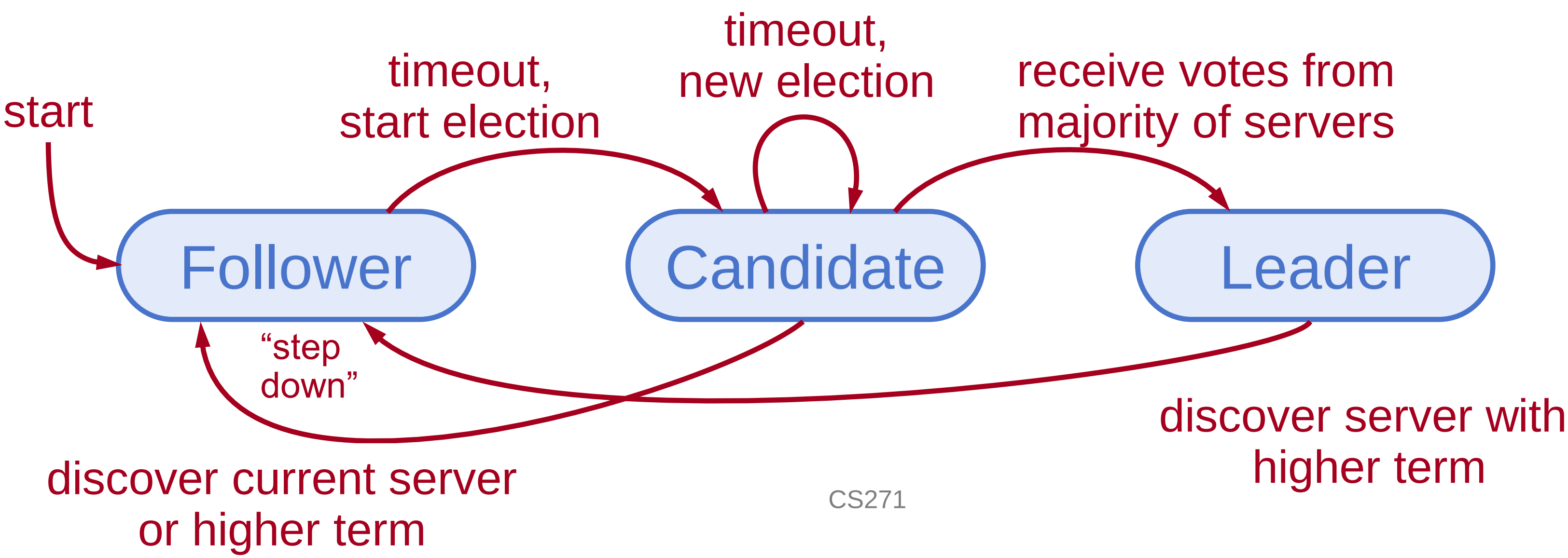
- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server
- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
- **Raft uses a leader:**
  - Decomposes the problem (normal operation, leader changes)
  - Simplifies normal operation (no conflicts)
  - More efficient than leader-less approaches

Raft Overview

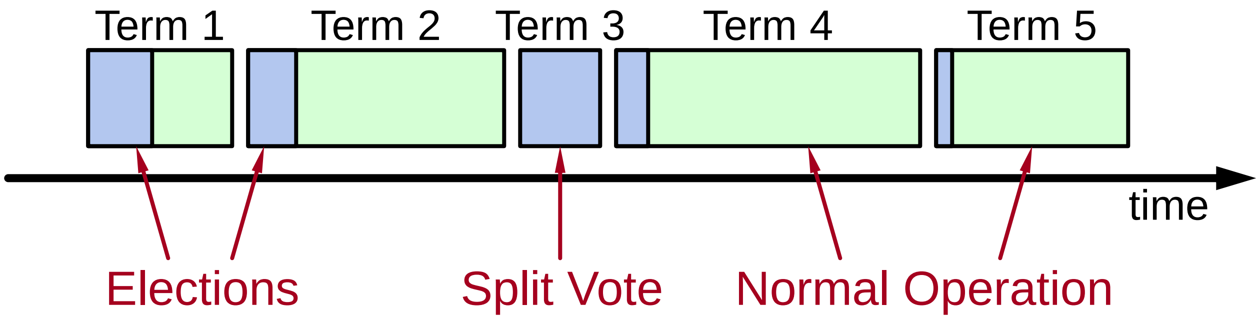
1. **Leader election:**
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. **Normal operation (basic log replication)**
3. **Safety and consistency after leader changes**
4. **Client interactions**
   - Implementing linearizeable semantics
5. **Configuration changes:**
   - Adding and removing servers

Server States

- **At any given time, each server is either:**
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
- **Normal operation: 1 leader, N-1 followers**



Terms



- **Time divided into terms (sometimes called epochs):**
  - **Election**
  - **Normal operation under a single leader**
- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value**
- **Key role of terms: identify obsolete information**

Raft Protocol Summary

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

Heartbeats and Timeouts

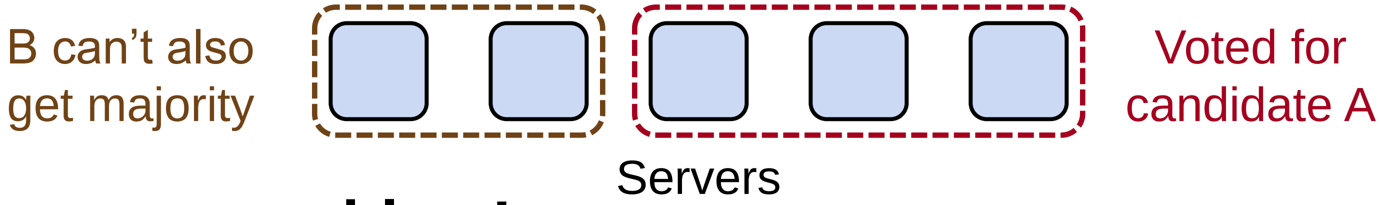
- **Servers start up as followers**
- **Followers expect to receive RPCs from leaders or candidates**
- **Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority**
- **If electionTimeout elapses with no RPCs:**
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Election Basics

- **Increment current term**
- **Change to Candidate state**
- **Vote for self**
- **Send RequestVote RPCs to all other servers, retry until either:**
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

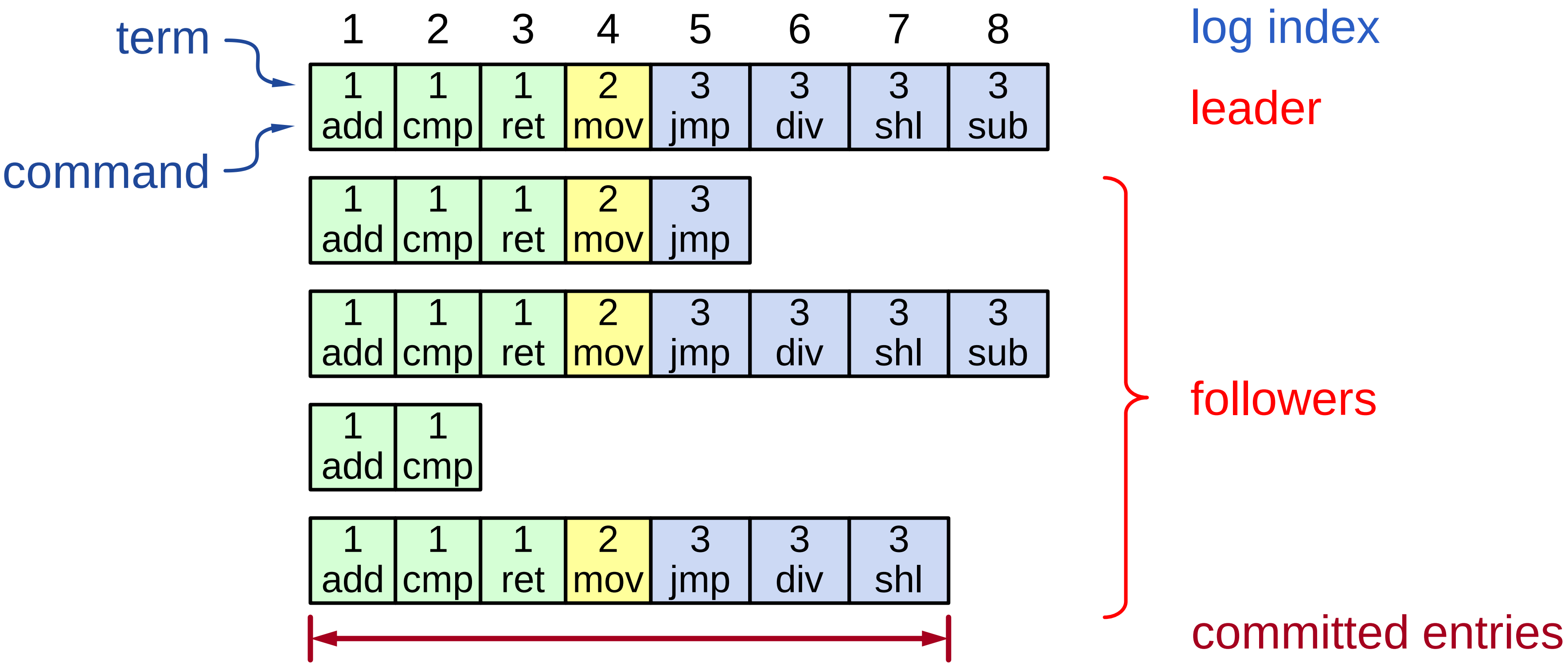
Elections, cont'd

- **Safety: allow at most one winner per term**
  - Each server gives out only one vote per term (persist on disk)
  - 2 different candidates can't accumulate majorities in same term



- **Liveness: problem!**
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time

Log Structure

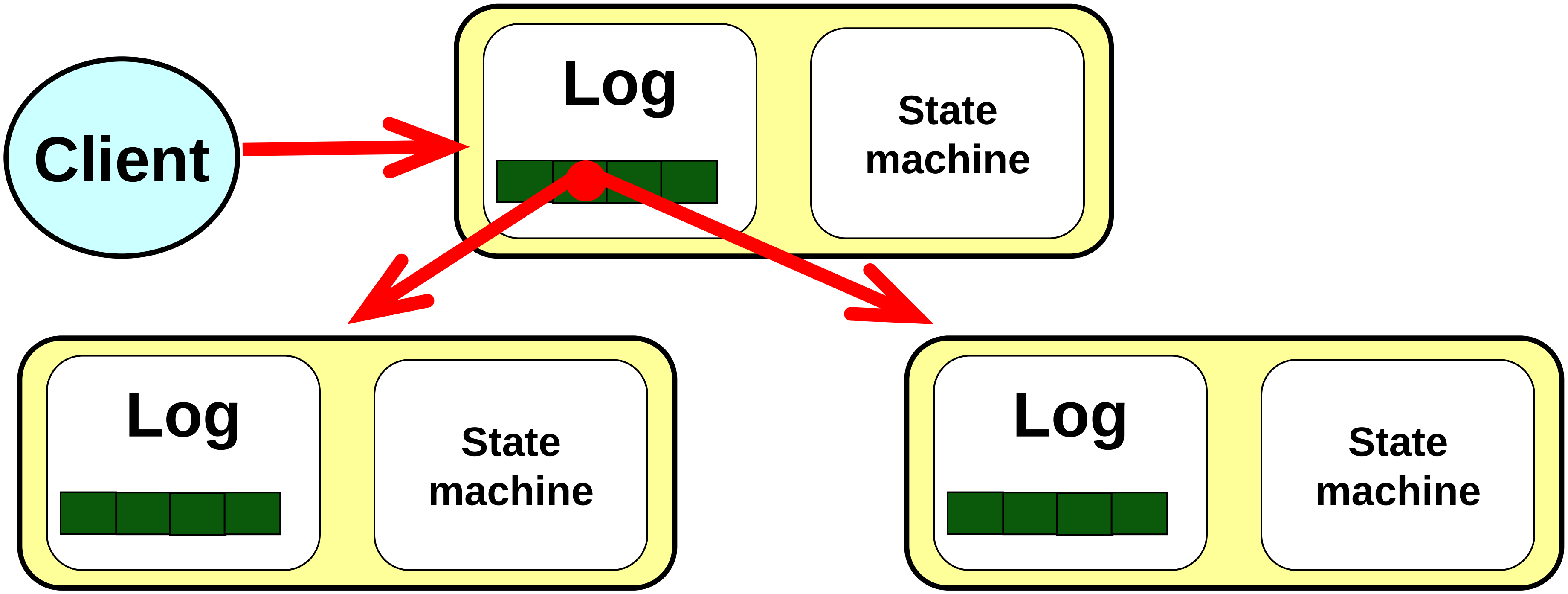


- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**
- **Entry committed if known to be stored on majority of servers**
  - Durable, will eventually be executed by state machines

1. Normal Operation

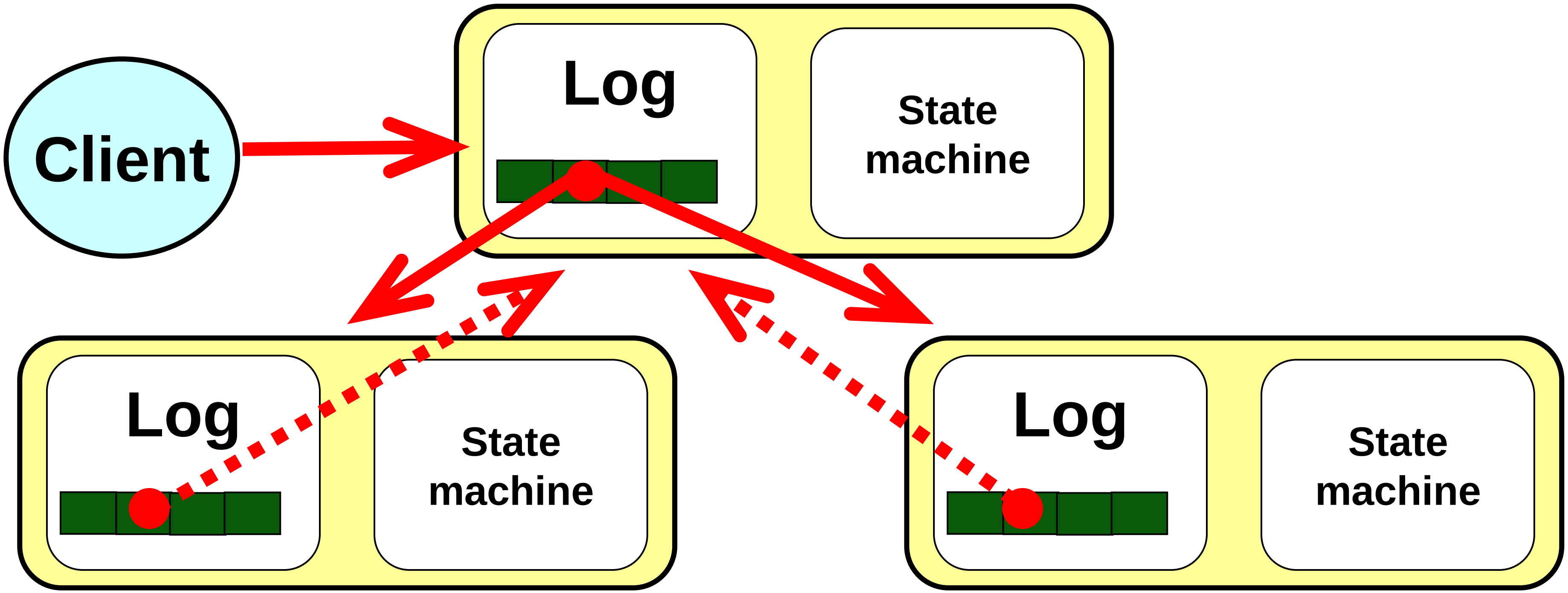
- **Client sends command to leader**
- **Leader appends command to its log**
- **Leader sends AppendEntries RPCs to followers**
- **Once new entry committed (heard from majority):**
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- **Crashed/slow followers?**
  - Leader retries RPCs until they succeed
- **Performance is optimal in common case:**
  - One successful RPC to any majority of servers

A client sends a request



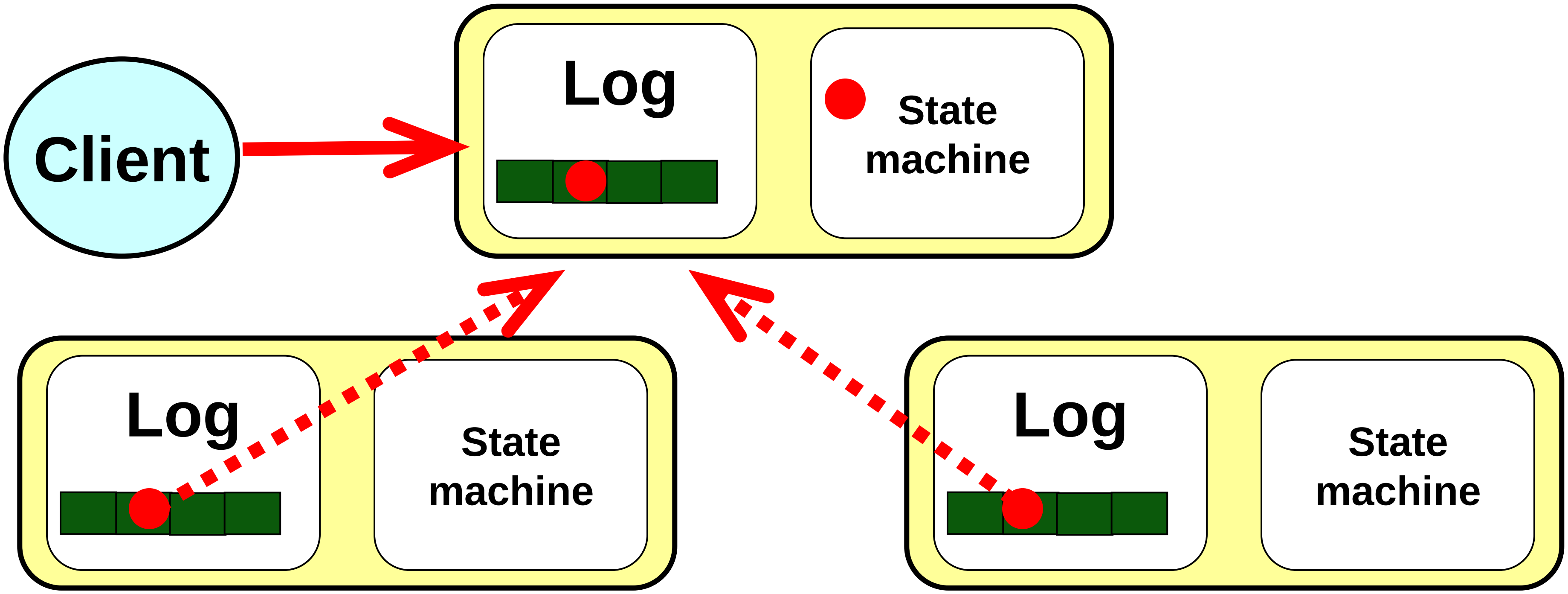
- **Leader stores request on its log and forwards it to its followers**

The followers receive the request



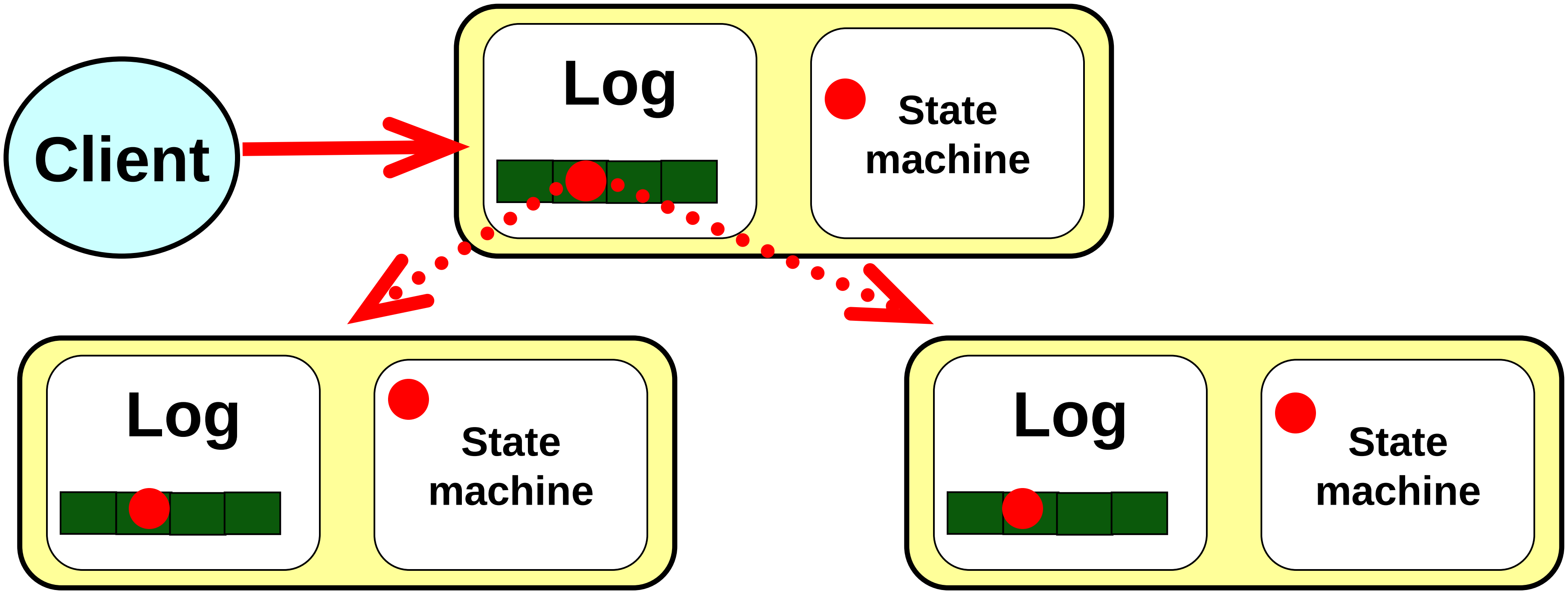
- **Followers store the request on their logs and acknowledge its receipt**

The leader tallies followers' ACKs



- **Once it ascertains the request has been processed by a majority of the servers, it updates its state machine**

The leader tallies followers' ACKs

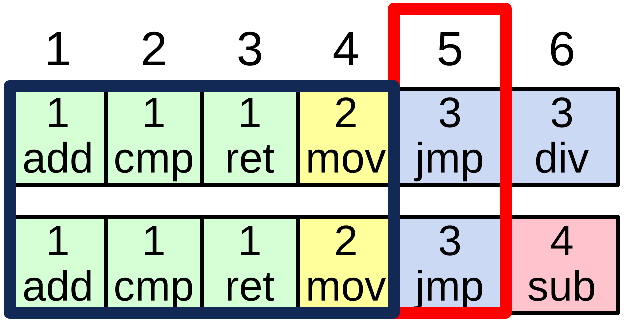


- **Leader's heartbeats convey the news to its followers: they update their state machines**

Log Consistency

High level of coherency between logs:

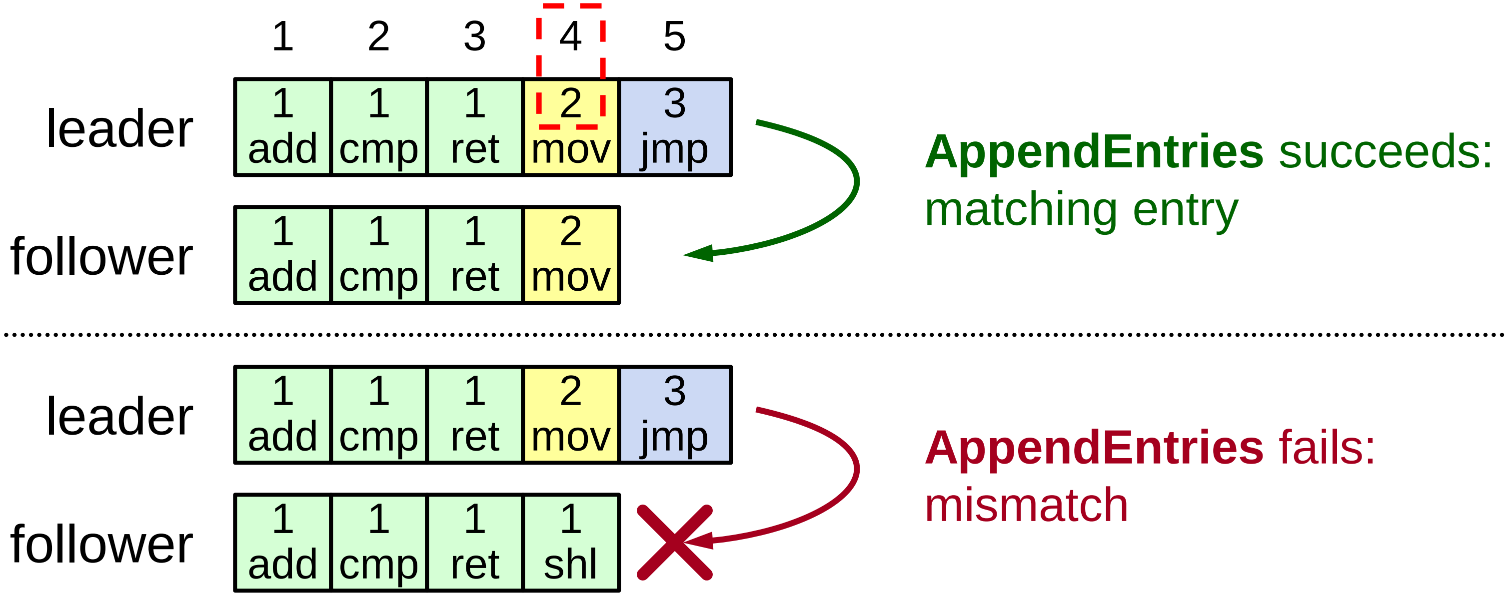
- **If log entries on different servers have same index and term:**
  - They store the same command
  - The logs are identical in all preceding entries



- **If a given entry is committed, all preceding entries are also committed**

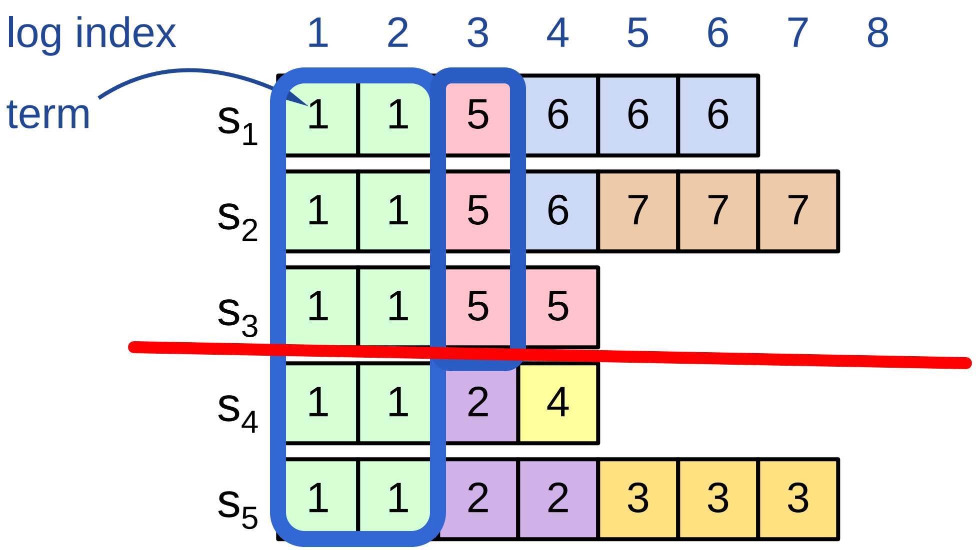
AppendEntries Consistency Check

- **Each AppendEntries RPC contains <index, term> of entry preceding new ones**
- **Follower must contain matching entry; otherwise it rejects request**
- **Implements an induction step, ensures coherency**



2. Leader Changes

- **At beginning of new leader's term:**
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous log entries:



Safety Requirement

Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- **Raft safety property:**
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
- **This guarantees the safety requirement**
  - Leaders never overwrite entries in their logs
  - Only entries in the leader's log can be committed
  - Entries must be committed before applying to state machine

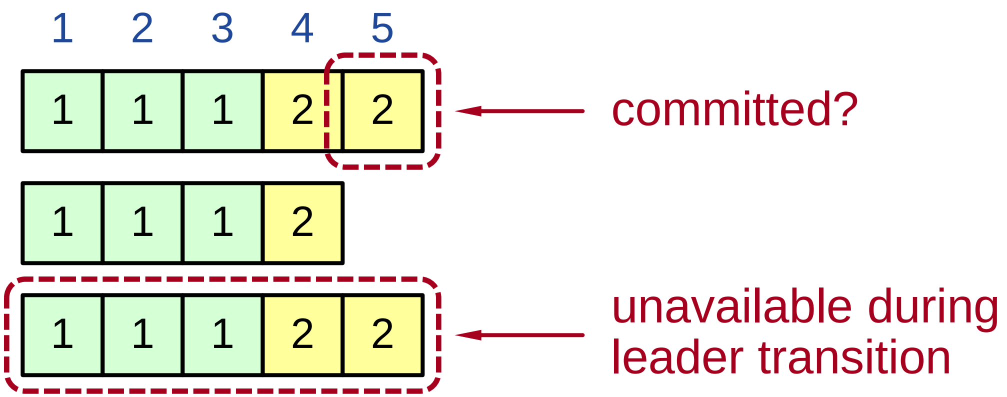
Committed → Present in future leaders' logs

Restrictions on commitment

Restrictions on leader election

Picking the Best Leader

- **Can't tell which entries are committed!**



- **During elections, choose candidate with log most likely to contain all committed entries**
  - Candidate C includes log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more complete":

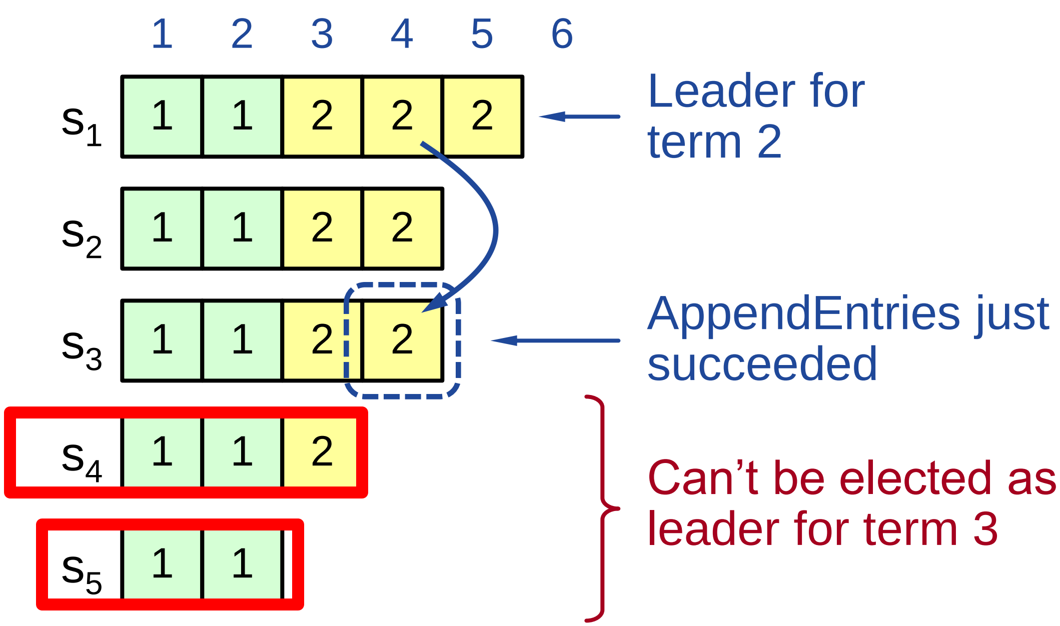
$(lastTerm_V > lastTerm_C) \ ||$

$(lastTerm_V == lastTerm_C) \ \&\& \ (lastIndex_V > lastIndex_C)$

  - Leader will have "most complete" log among electing majority

Committing Entry from Current Term

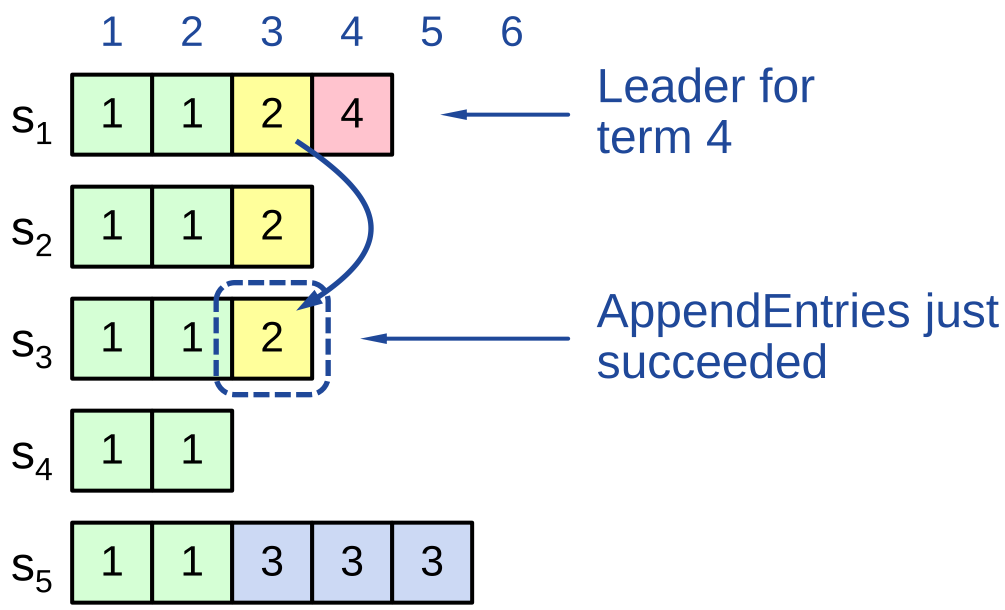
- **Case #1/2: Leader decides entry in current term is committed**



- **Safe: leader for term 3 must contain entry 4**

Committing Entry from Earlier Term

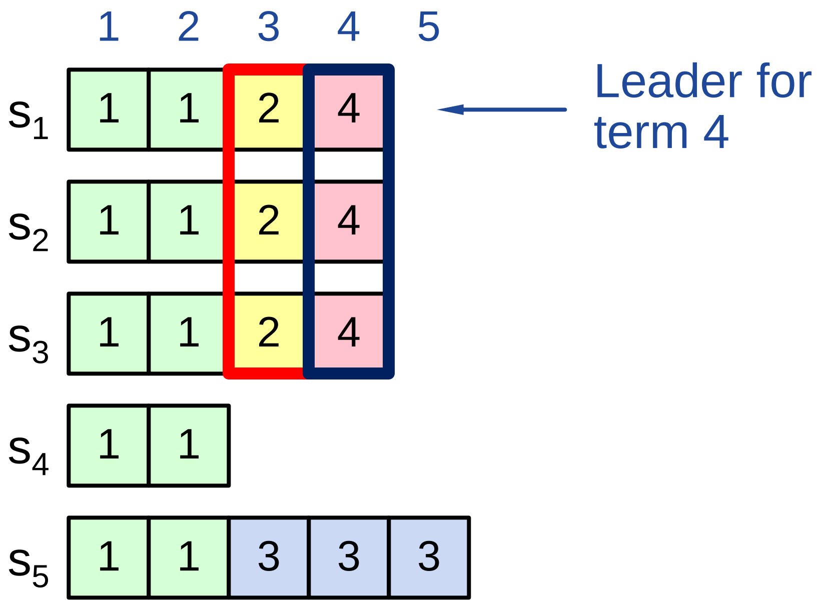
- **Case #2/2: Leader is trying to finish committing entry from an earlier term**



- **Entry 3 not safely committed:**
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

New Commitment Rules

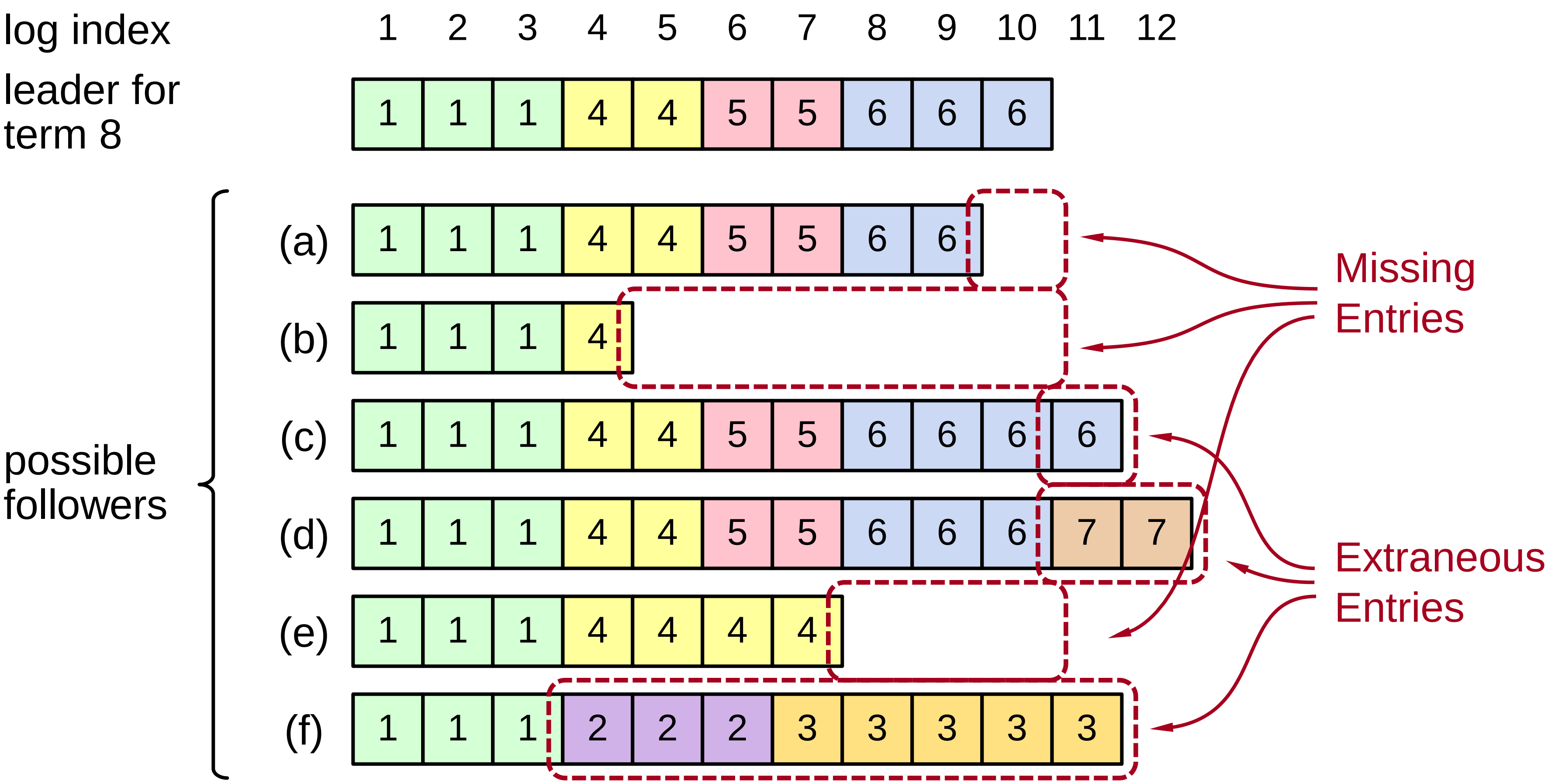
- **For a leader to decide an entry is committed:**
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- **Once entry 4 committed:**
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

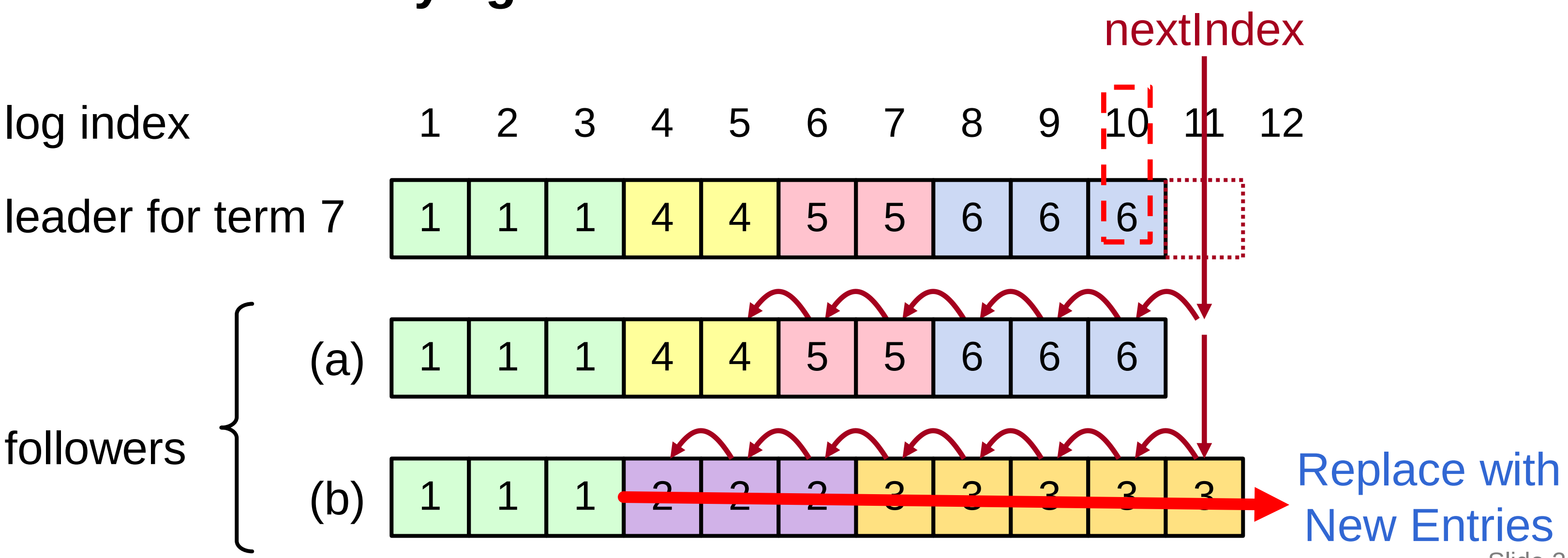
Log Inconsistencies

Leader changes can result in log inconsistencies:



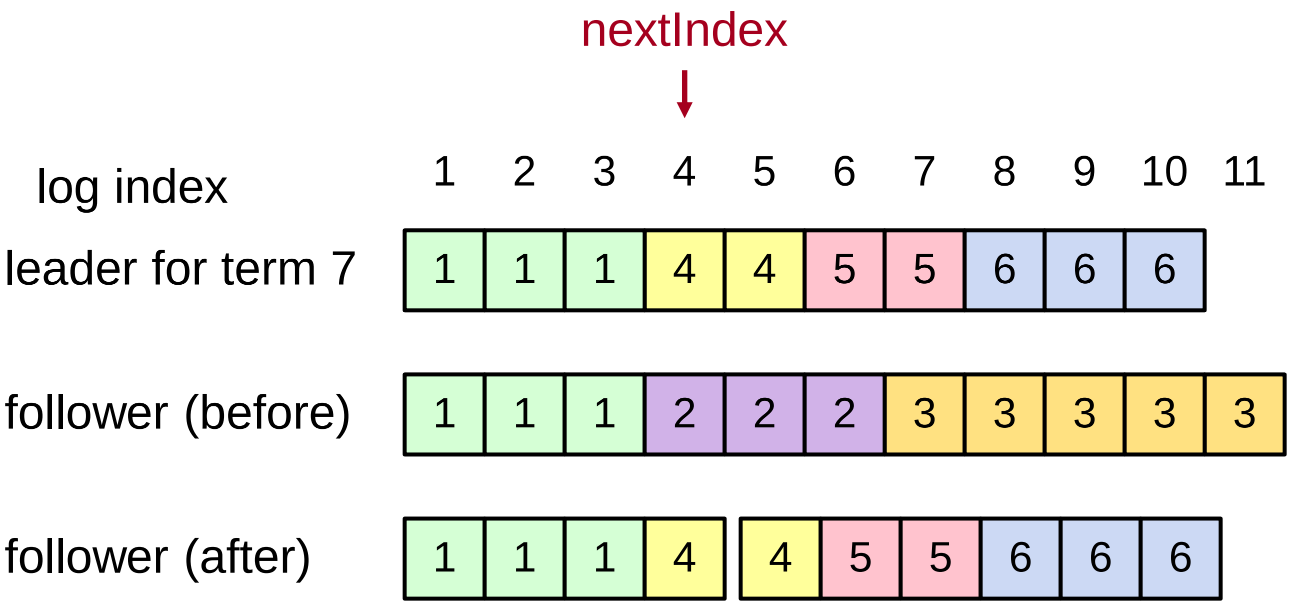
Repairing Follower Logs

- **New leader must make** **follower logs consistent with its own**
  - Delete extraneous entries
  - Fill in missing entries
- **Leader keeps** **nextIndex** **for each follower:**
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- **When AppendEntries consistency check fails, decrement nextIndex and try again:**



Repairing Logs, cont’d

- **When follower overwrites inconsistent entry, it deletes all subsequent entries:**



Neutralizing Old Leaders

- **Deposed leader may not be dead:**
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- **Terms used to detect stale leaders (and candidates)**
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- **Election updates terms of majority of servers**
  - Deposed server cannot commit new log entries

3. Client Protocol

- **Send commands to leader**
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
- **Leader does not respond until command has been logged, committed, and executed by leader’s state machine**
- **If request times out (e.g., leader crash):**
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol, cont’d

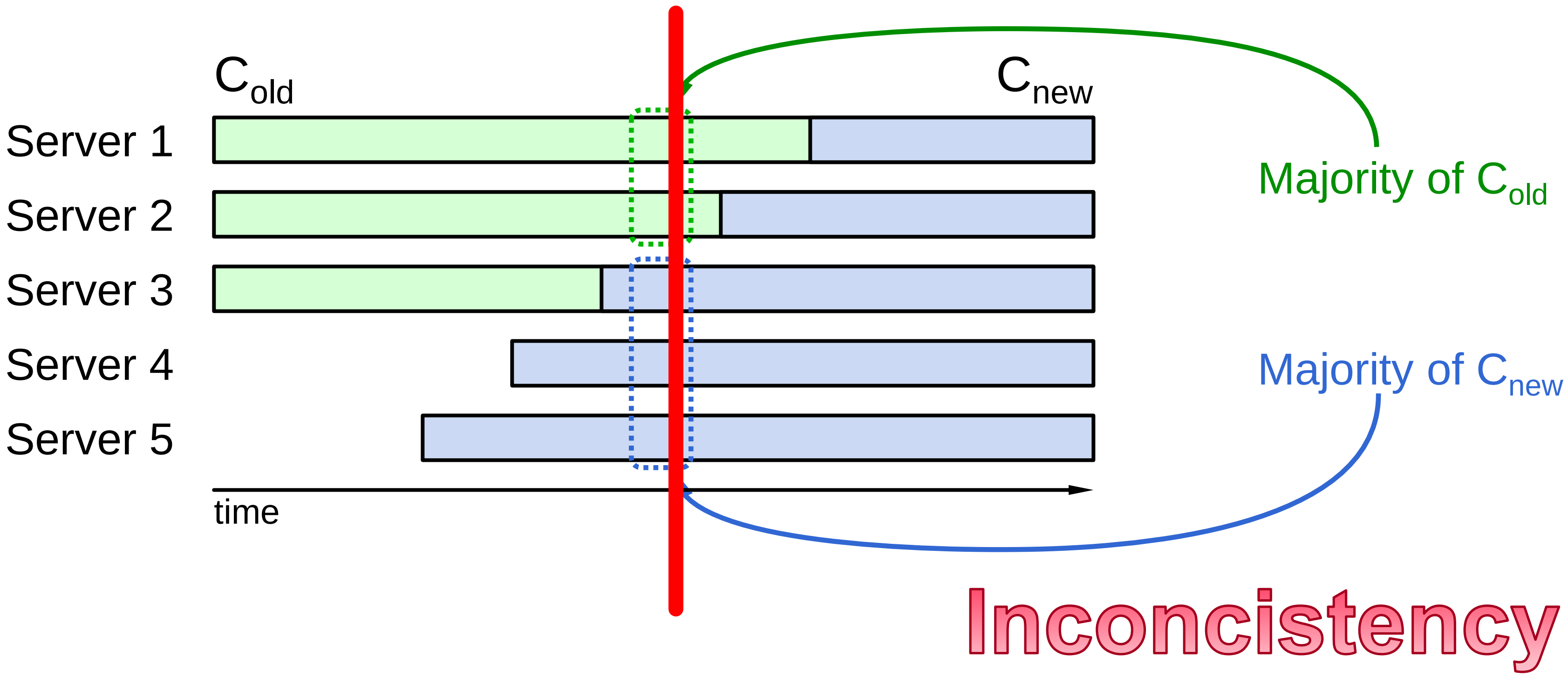
- **What if leader crashes after executing command, but before responding?**
  - Must not execute command twice
- **Solution: client embeds a unique id in each command**
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command
- **Result: exactly-once semantics as long as client doesn’t crash**

4. Configuration Changes

- **System configuration:**
  - ID, address for each server
  - Determines what constitutes a majority
- **Consensus mechanism must support changes in the configuration:**
  - Replace failed machine
  - Change degree of replication

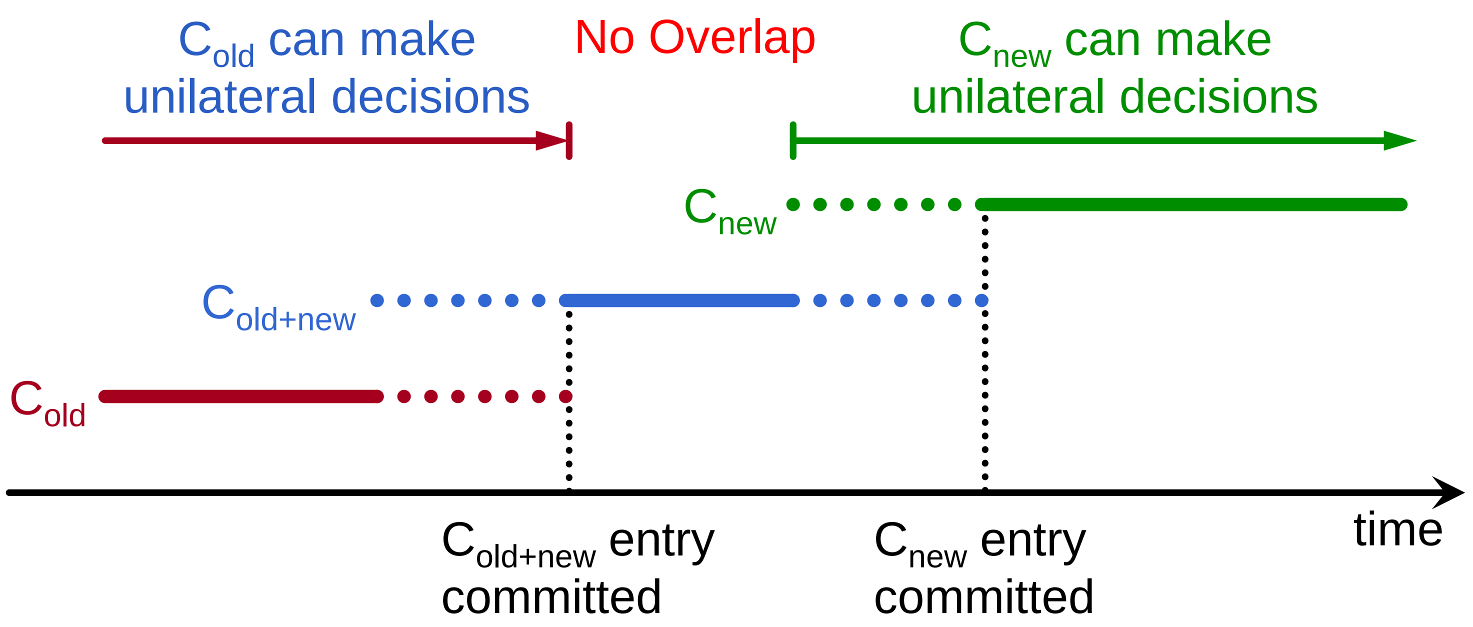
Configuration Changes, cont'd

Cannot switch directly from one configuration to another: conflicting majorities could arise



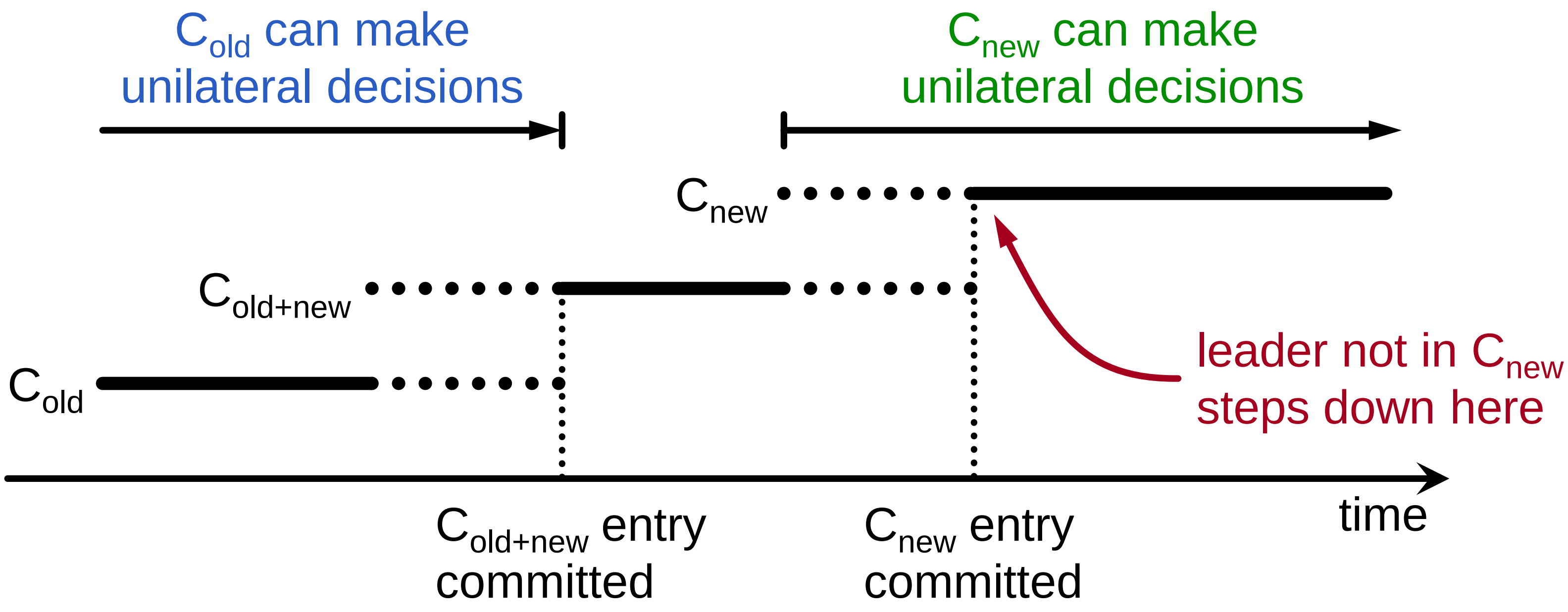
Joint Consensus

- **Raft uses a 2-phase approach:**
  - Intermediate phase uses joint consensus (need majority of **both** old and new configurations for elections, commitment)
  - Configuration change is just a log entry; **applied immediately** on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration



Joint Consensus, cont'd

- **Additional details:**
  - Any server from either configuration can serve as leader
  - If current leader is not in C_{new}, must step down once C_{new} is committed.



Raft Summary

1. Leader election
2. Normal operation
3. Configuration changes