Raft: A Consensus Algorithm for Replicated Logs

Slides from Diego Ongaro and John Ousterhout, Stanford University

Announcements

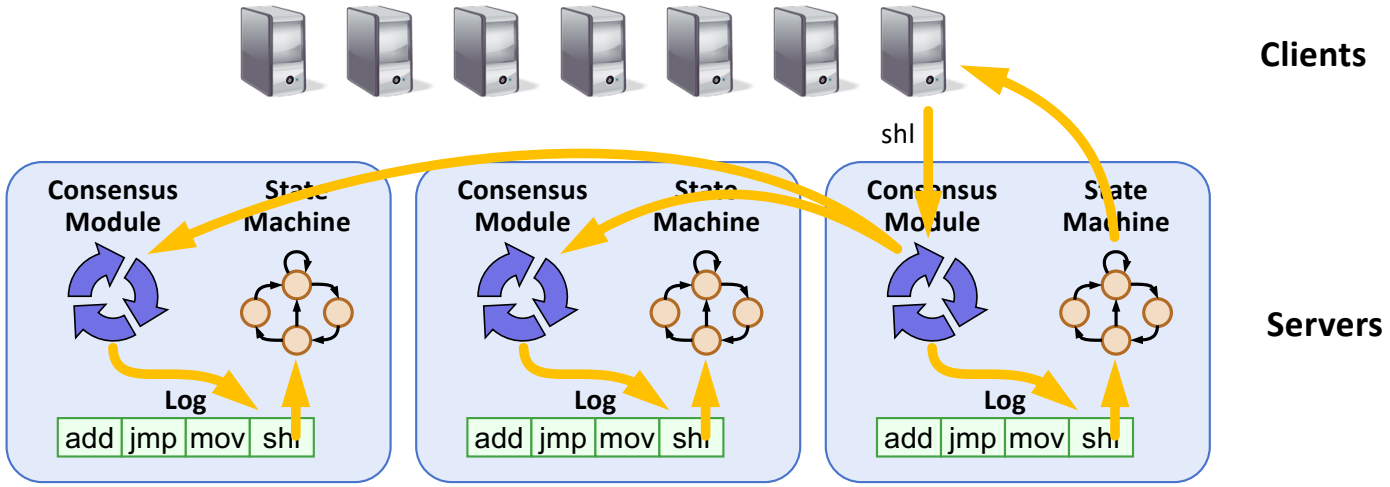
HW3 out, due Oct 29

Midterm 2 on Nov 1

- Covers leader election and consensus
- Including sync consensus, FLP, Paxos, Raft
- Not including Bitcoin

MP2 out shortly, implement Raft

Goal: Replicated Log



- Replicated log => replicated state machine
  - All servers execute same commands in same order
- 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

Goal: Design for understandability

- Main objective of Raft's design
  - Whenever possible, select the alternative that is the easiest to understand.

- Techniques that were used include
  - Dividing problems into smaller problems.
  - Reducing the number of system states to consider.

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. Neutralizing old leaders

Raft Overview

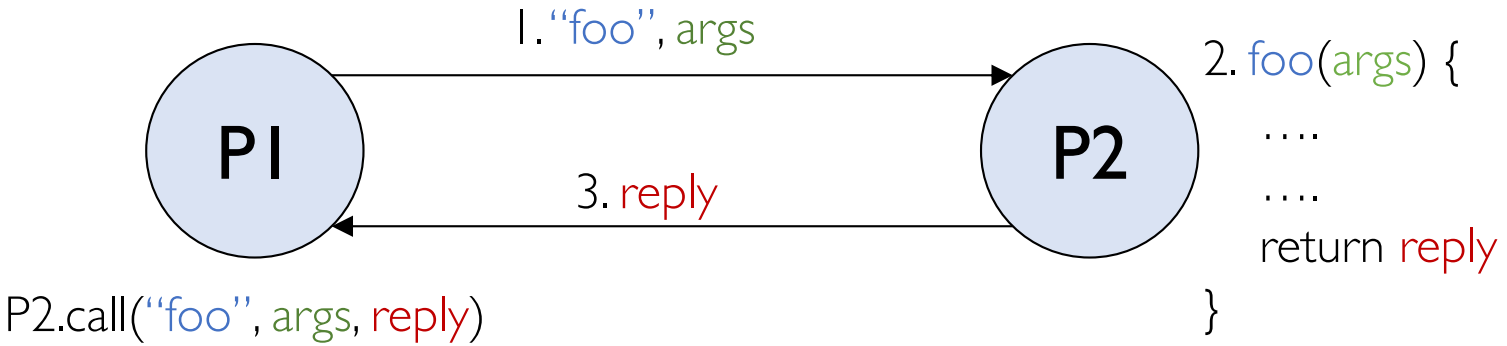
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders

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 (requests), responds to incoming RPCs
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

Quick Detour: RPCs

- Raft servers communicate via RPCs.
- What are RPCs?
  - Remote Procedure Calls: *procedure call between functions on different processes*
  - *Convenient programming abstraction.*



Raft Election

To start an election, candidate sends RequestVote RPCs to all other processes

When receiving a RequestVote RPC, process votes yes unless it has already voted for someone else

Candidate declares self leader after receiving *majority* of votes (including from itself)

Raft Election: Terms

Split vote: no leader elected

After electionTimeout, increment term, restart election

electionTimeout is *randomized* to reduce chance of multiple people starting election at the same time

Raft Election: Heartbeats

Once leader is elected leader (majority votes), it starts transmitting heartbeats including term #

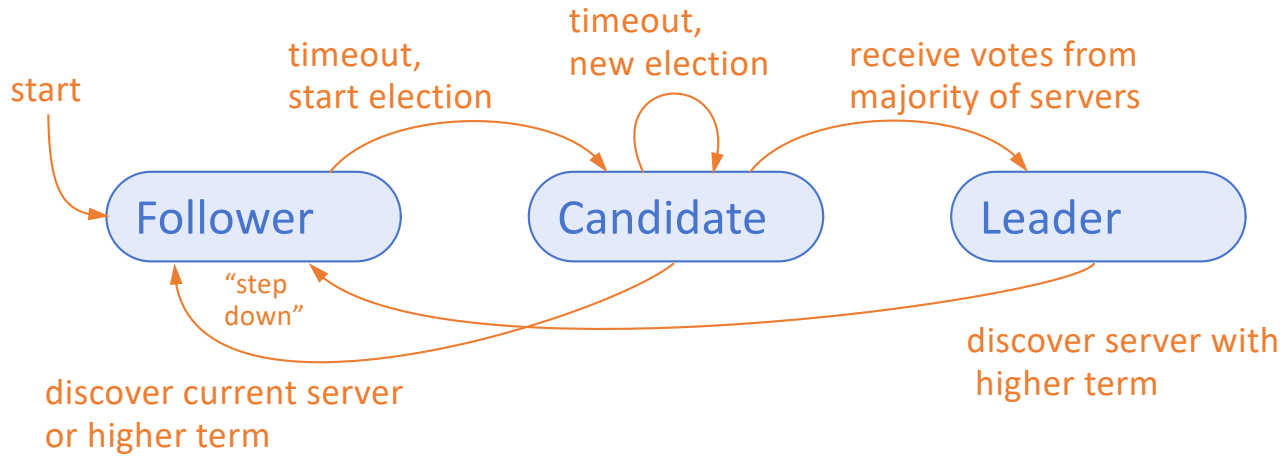
Receiving a heartbeat puts other processes into follower state

Followers set timeout; if no heartbeat within timeout, start election

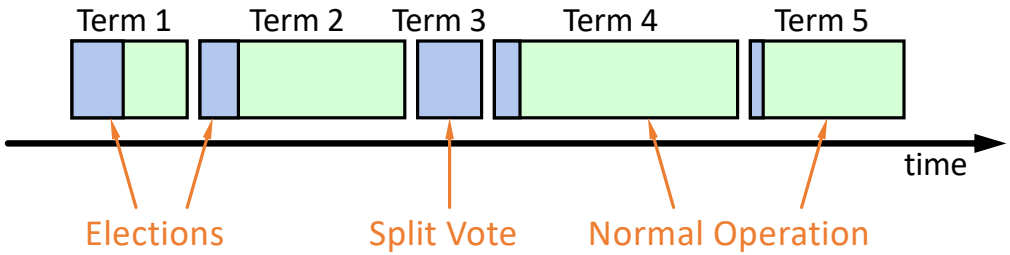
Again, randomized timeouts minimize chance of split vote

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:
  - 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

Heartbeats and Timeouts

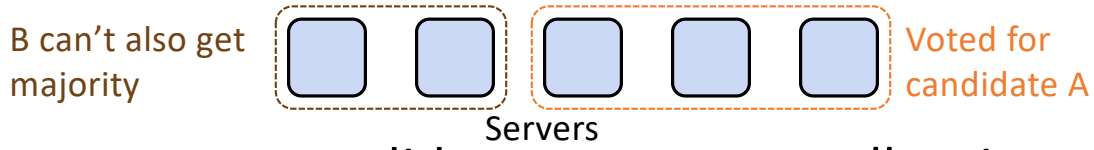
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts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

Election Basics

- On timeout:
  - Increment current term
  - Change to Candidate state
  - Vote for self
  - Send RequestVote RPCs to all other servers:
    1. Receive votes from majority of servers:
       - Become leader
       - Send AppendEntries heartbeats (RPCs) 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)
  - Two different candidates can't accumulate majorities *in same term*



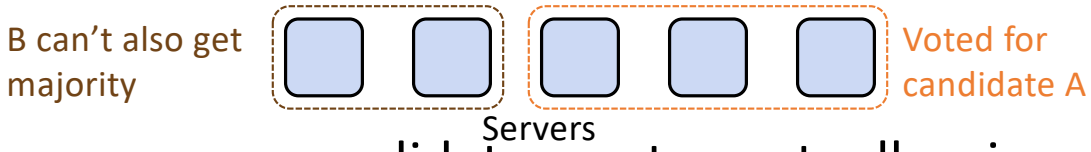
- Liveness: some candidate must eventually win

Safety is guaranteed. Liveness is not.

- *Election may result in a split vote – no candidate gets majority.*

Elections, cont'd

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

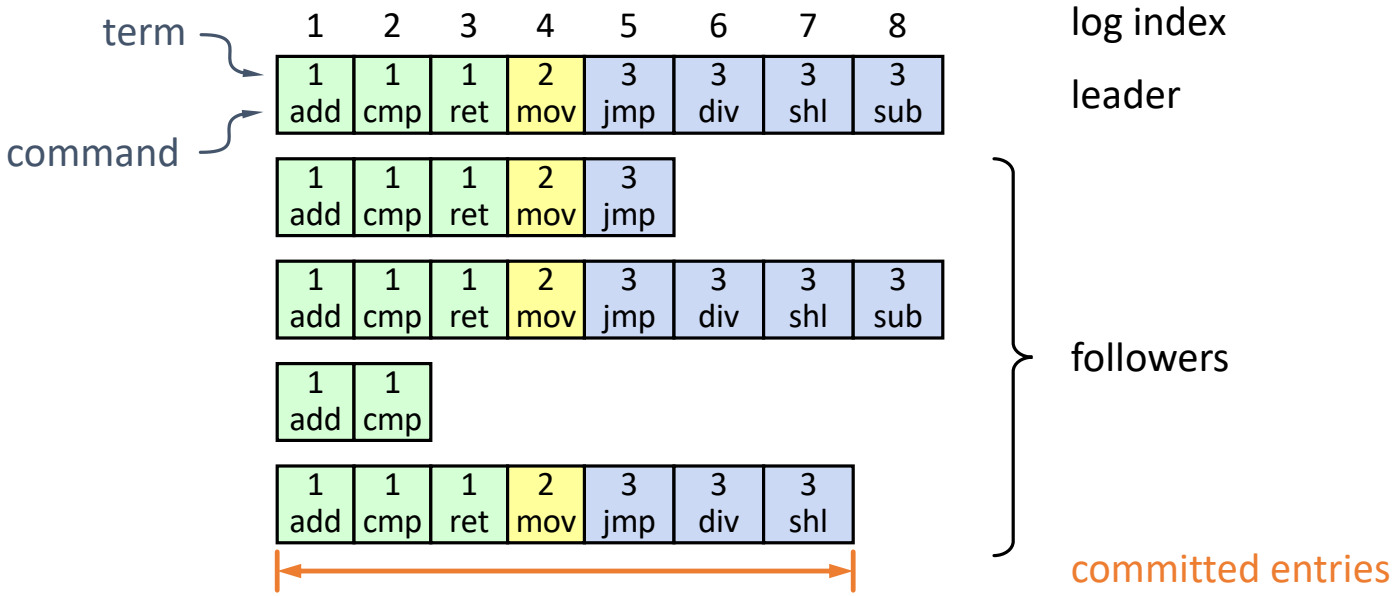


- Liveness: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time
- *Safety is guaranteed. Liveness is not.*
  - *Election may result in a split vote – no candidate gets majority.*

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. Neutralizing old leaders

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

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, 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

Today

- Finish Raft
  - Log consistency
- Bitcoin / Nakamoto Consensus

Announcements

Midterm 2: Monday, Nov 1, 7-8:30 p.m.

- Material covered: leader election and consensus, up to and including Raft
- Midterm will be on Prairielearn
- Optional review session *online only* on Thu, Oct 28
- Conflict form going up soon

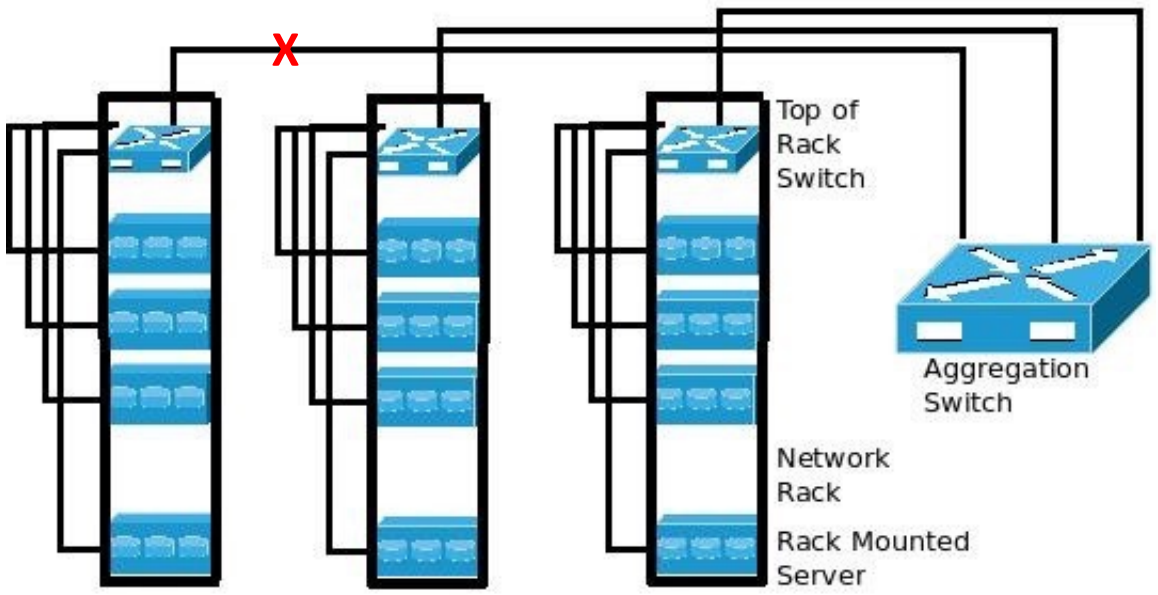
MP2: Released today

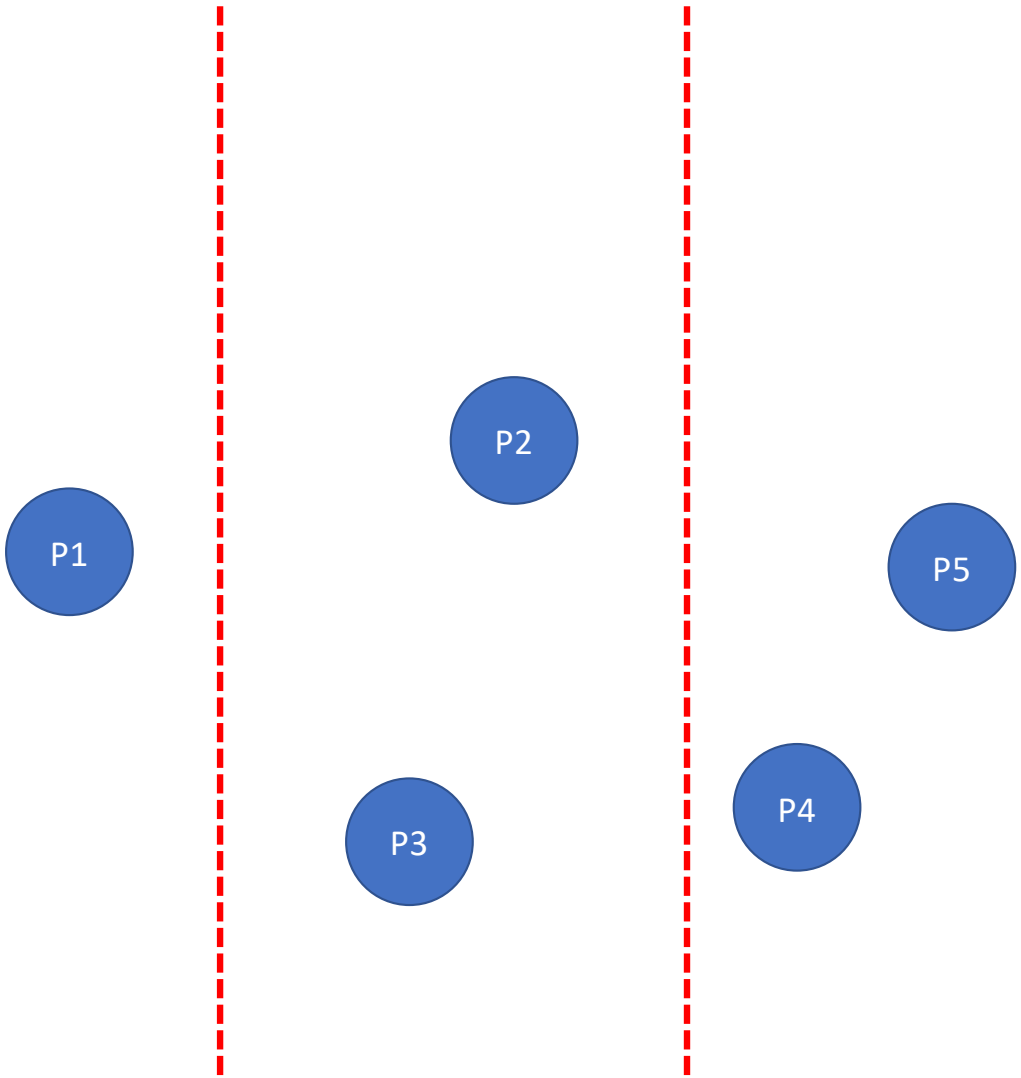
- Implement Raft functionality

Raft Review

- Leader election:
  - Safety: each term has *at most* one leader
  - Liveness: If no leader, start election for next term after a timeout
- Normal operations:
  - Events sent to leader
  - Leader adds events to log, replicates to followers
  - Event committed after a majority of followers acknowledge

Partitions

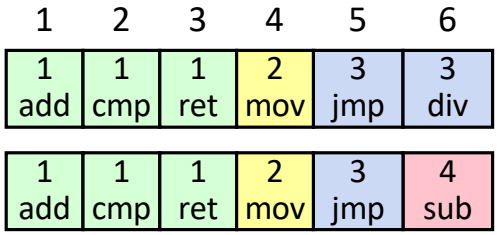

X
Top of
Rack
Switch
Aggregation
Switch
Network
Rack
Rack Mounted
Server


P1
P2
P3
P4
P5

Log Consistency

High level of coherency between logs:

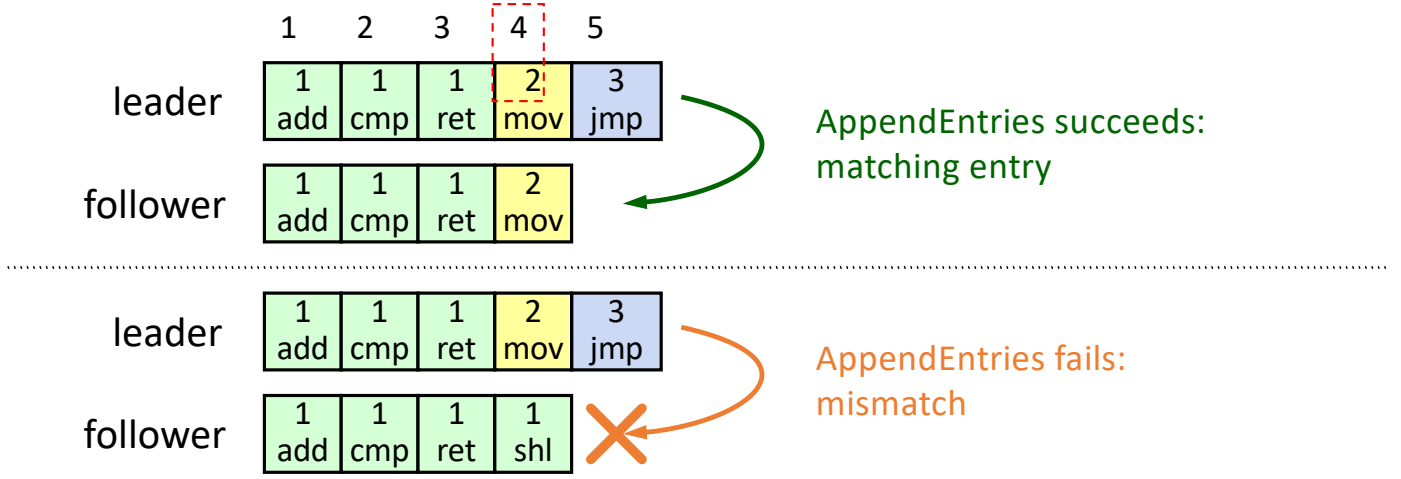
- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries



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

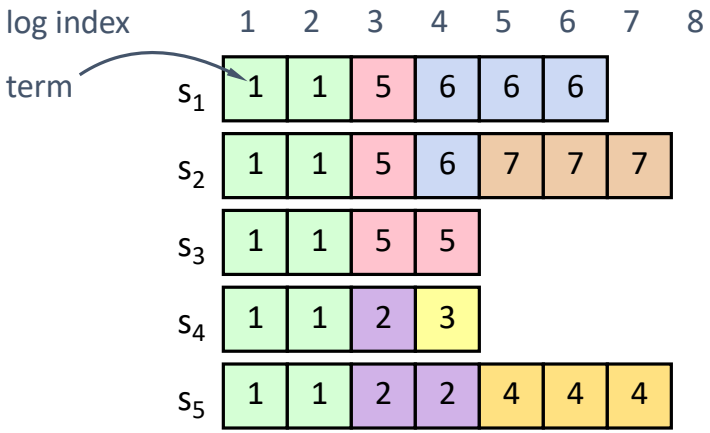
AppendEntries Consistency Check

- 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



Leader Changes

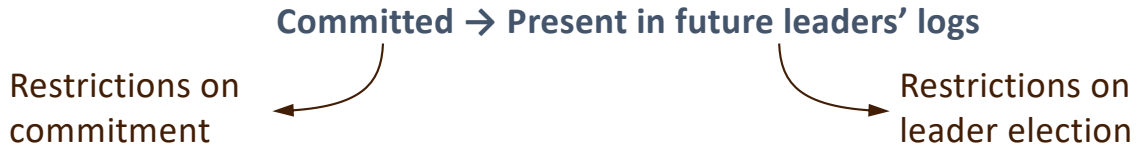
- At beginning of new leader's term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous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



Safety Requirement

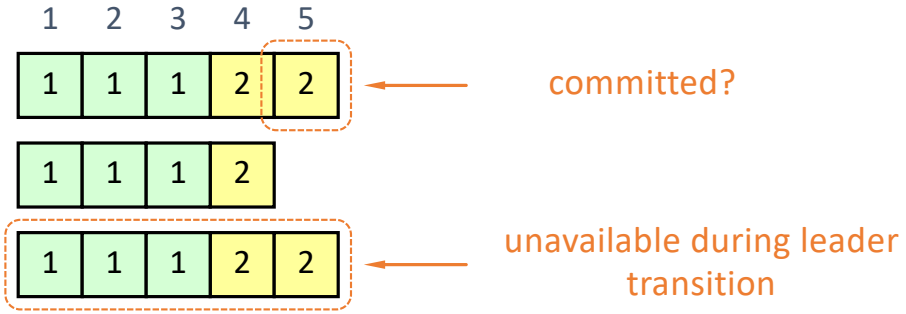
Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- 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



Picking the Best Leader

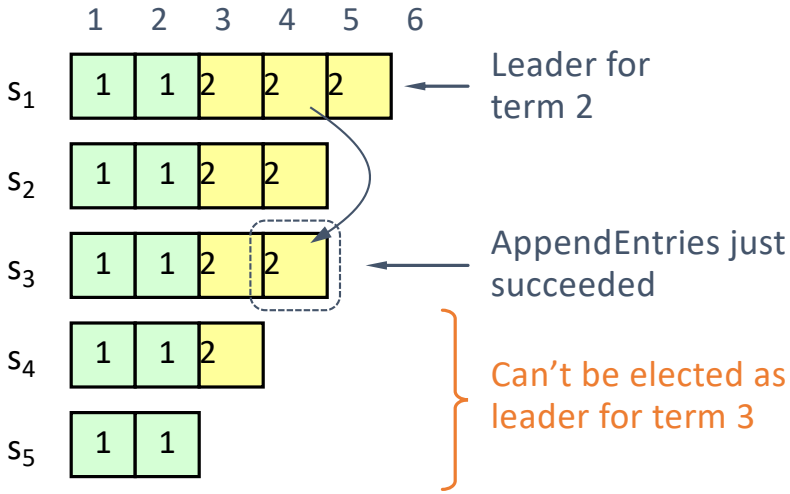
- Can't tell which entries are committed!



- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include 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

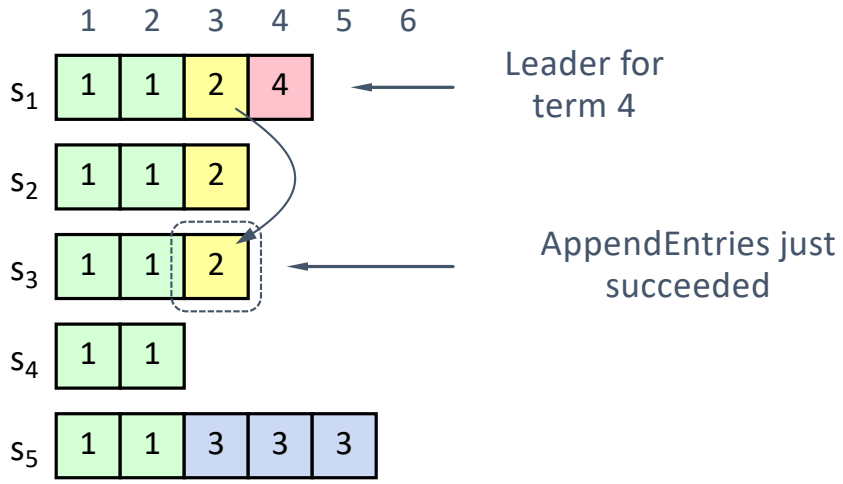
- Case #1/2: Leader decides entry in current term is committed



- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

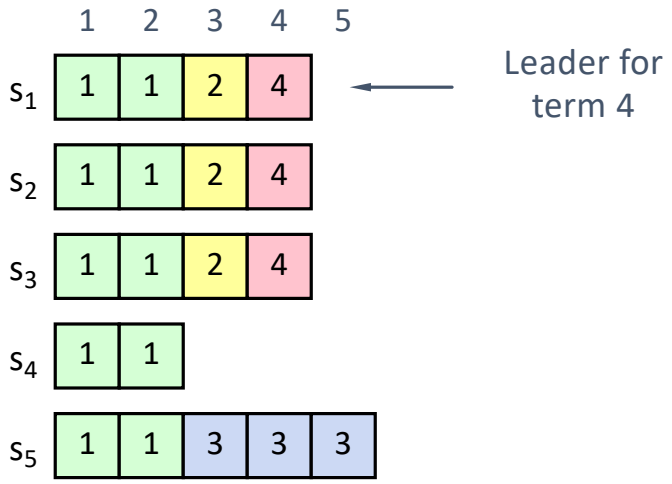
- Case #2/2: Leader is trying to finish committing entry from an earlier term



- 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

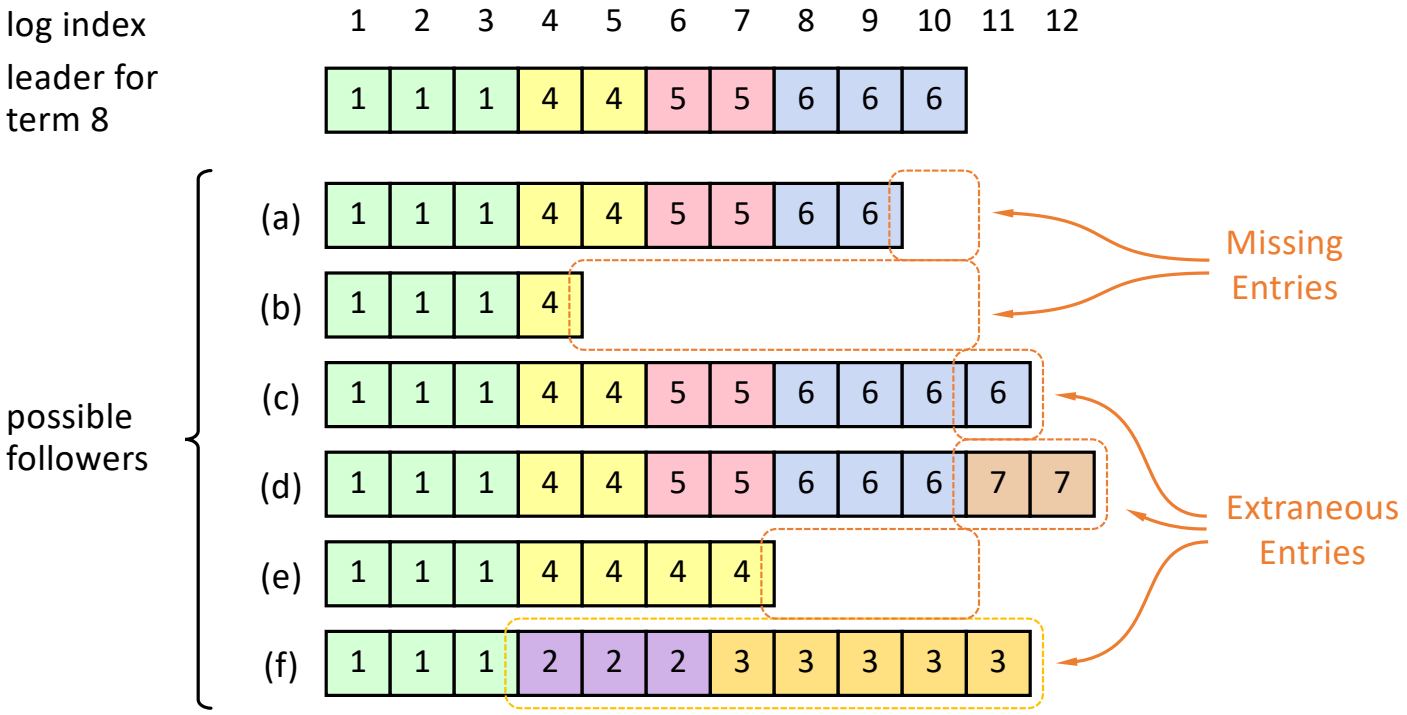
- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe.

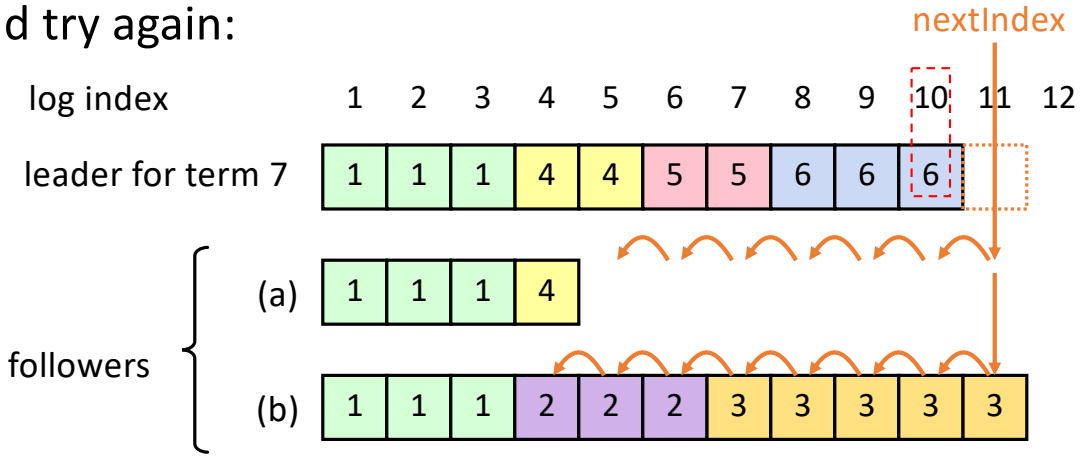
Log Inconsistencies

Leader changes can result in log inconsistencies:



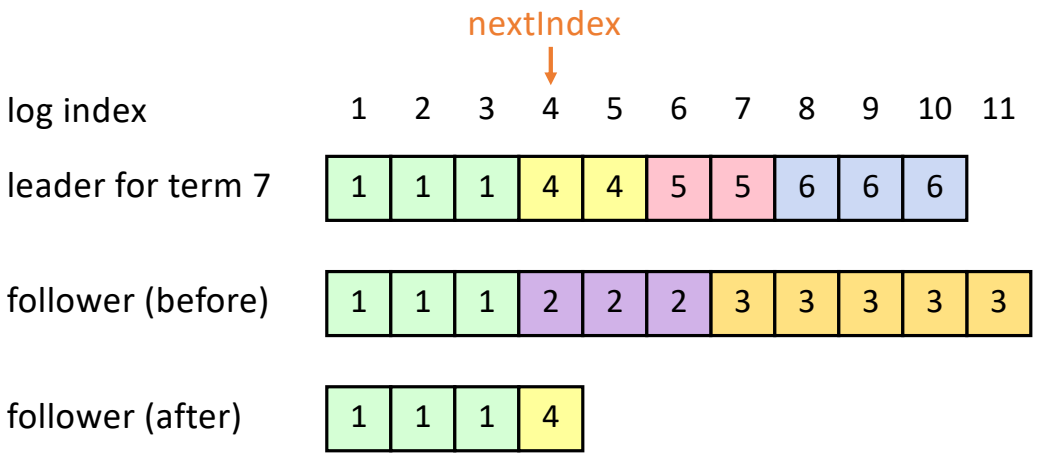
Repairing Follower Logs

- 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

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