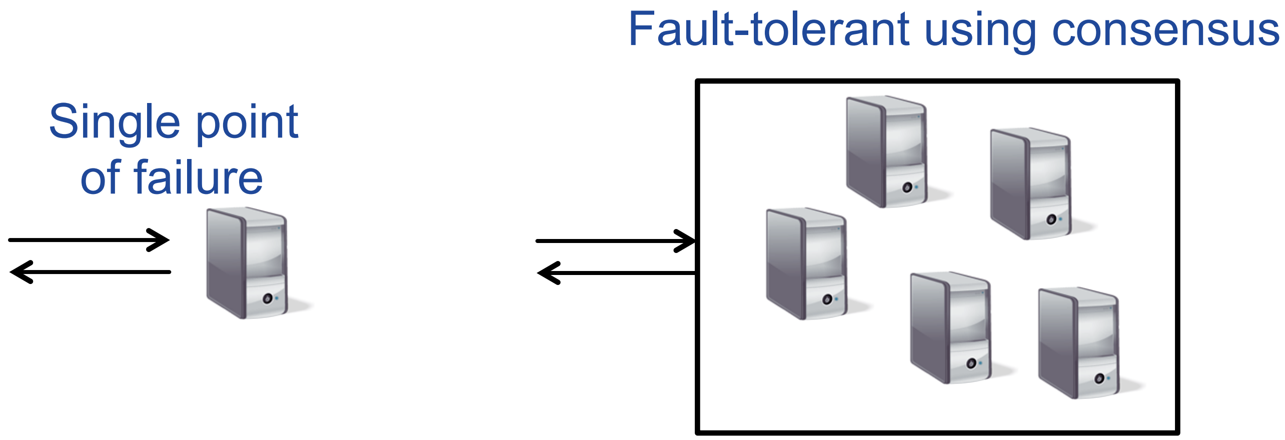
Raft: A Consensus Algorithm for Building Real Systems

Diego Ongaro and John Ousterhout

Stanford University

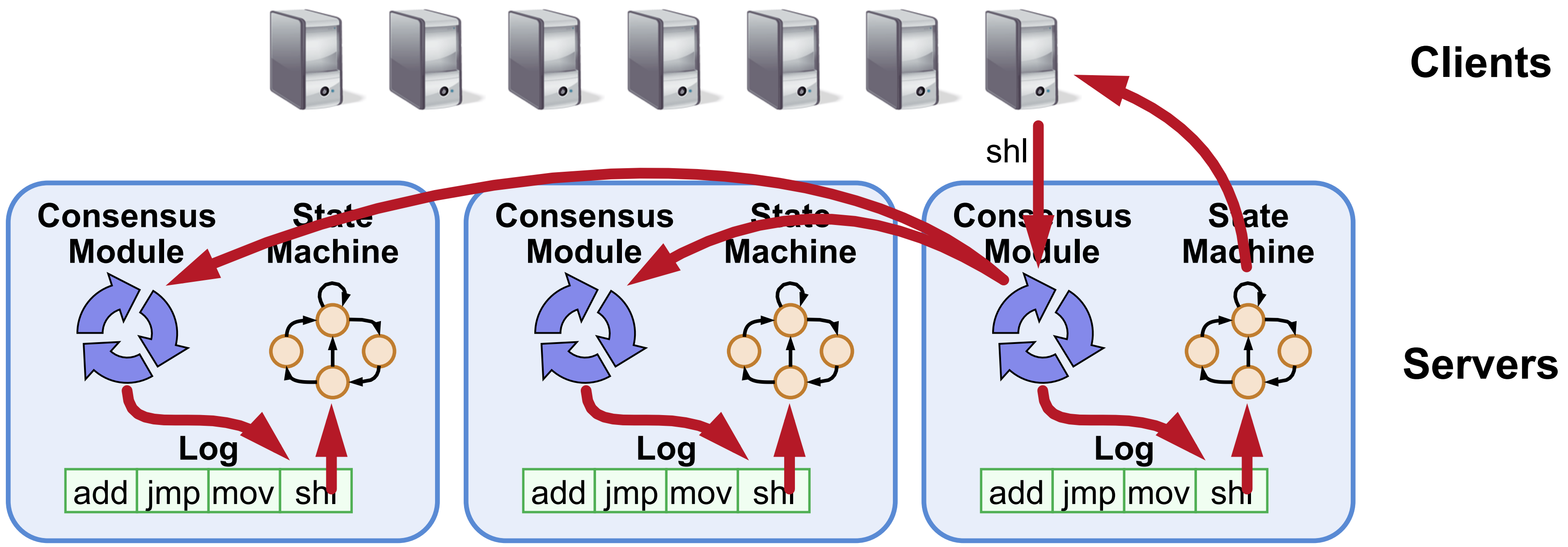
What’s consensus for?

- **Consensus algorithms can make any deterministic service fault-tolerant**



- **How is consensus used in practice?**
  - What are some systems that use consensus?
- **Why should you care?**

Goal: Replicated Log



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

How many faults can we handle?

- Assume servers fail by stopping (non-Byzantine)
- Let's say we have 5 servers:

- Answer: can tolerate 2 faults
- In general, a majority of servers is needed to ensure some overlap

Understandability

- **Our goal for Raft was to make it easy to understand**
- **How do you make a distributed system easy to understand?**
  - How is the web easy to understand?

- **Some ideas:**
  - Have clear sources of responsibility and truth
  - Split the problem into simpler parts
  - Reduce the number of cases to consider (state space complexity)

Leader-Oriented Approach

Two general approaches to consensus:

- **Symmetric, leader-less approach:**
  - All servers have equal roles
  - Clients can contact any server
- **Leader-oriented approach:**
  - 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)
  - More efficient than leader-less approaches
- **Which approach does Paxos use?**

Raft Overview

1. **Leader election:**
   - Selects one of the servers to act as leader
   - Detects crashes, chooses new leader
2. **Normal operation:**
   - Leader accepts client requests
   - Leader blindly overwrites other servers' logs with its own
   - Leader marks entries *committed* when they are safe to apply to state machines
3. **Safety**
   - Rigs leader election to ensure leader's log is "the truth"
   - Defines which entries may be marked committed

When should we elect a new leader?

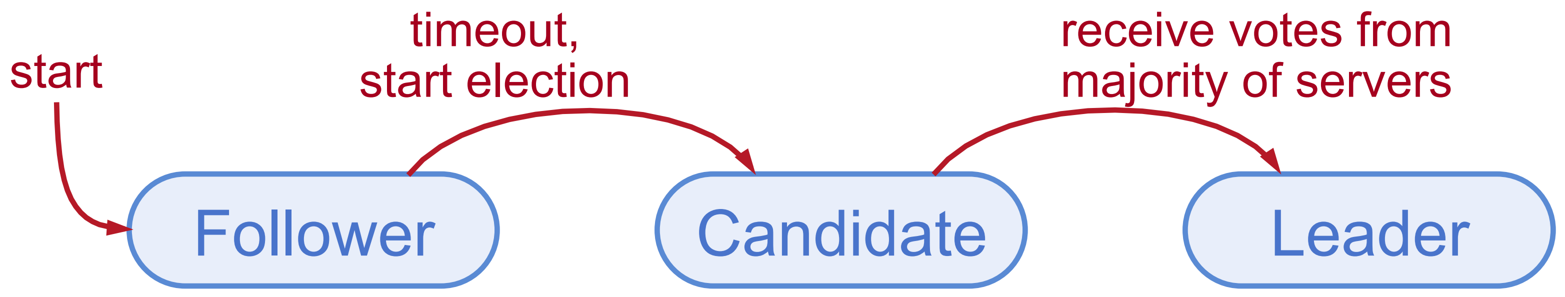
- **Servers start up as followers**
- **Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority**
- **Followers expect to receive these**
- **If electionTimeout elapses with no RPCs:**
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically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

How can we elect a new leader?

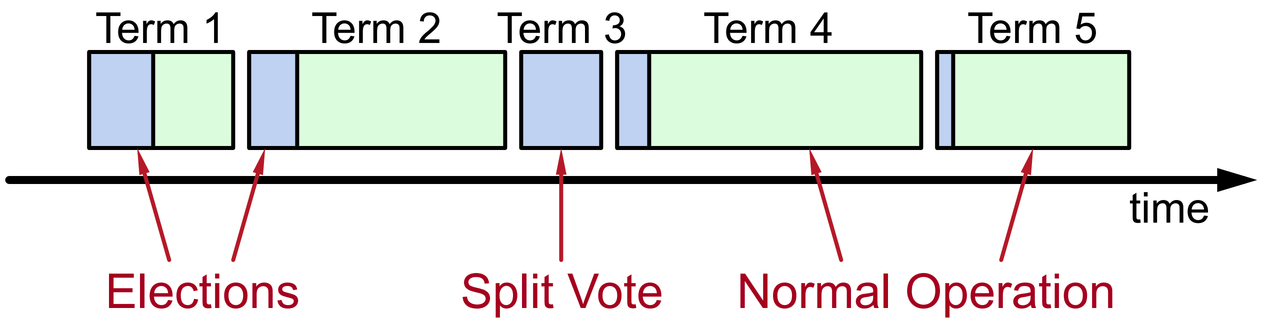
- **Vote on it**
  - Each server may cast one vote
  - Winner needs majority
- **What happens if we get stuck (split vote)?**

Server States

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



Terms



- **Time is divided into terms:**
  - Election
  - Normal operation under a single leader
- **Each server may grant one vote per term**
- **At most 1 leader per term**
- **Some terms have no leader (failed election)**

Terms Identify Obsolete Information

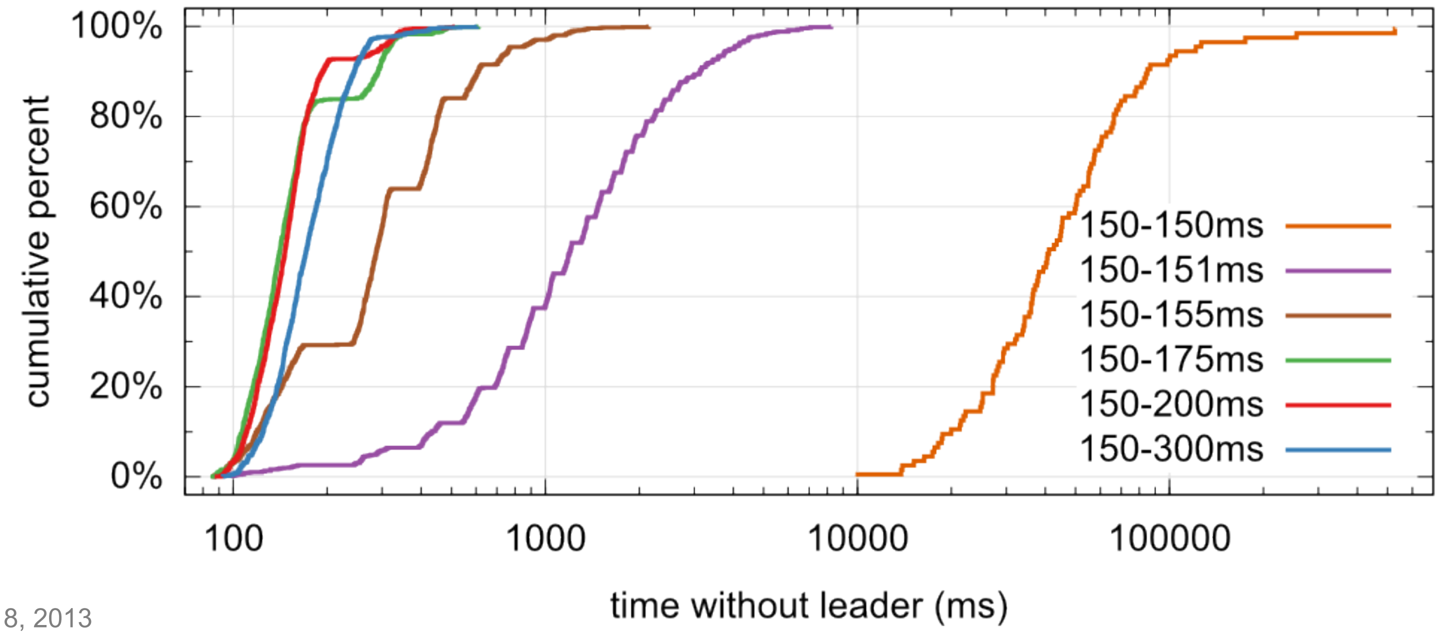
- **Each server maintains current term value**
- **Terms are propagated across RPCs**
  - Every RPC contains term of sender
  - If sender’s term is stale, RPC is rejected, sender reverts to follower and updates its term
  - If receiver’s term is stale, it reverts to follower, updates its term, then processes RPC normally
- **Terms detect stale candidates and leaders**
  - Prevent them from doing any harm

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. Election gets stuck:
     - Increment term, start new election

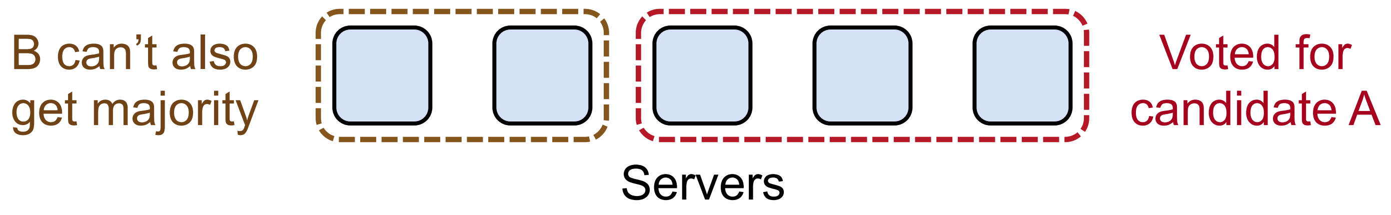
How do we detect stuck elections?

- **Seems tricky**
- **Simplest way: if another election timeout elapses without a winner, it's probably stuck**
- **What if the next election also gets stuck?**
  - Randomize the election timeouts



Election Summary

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



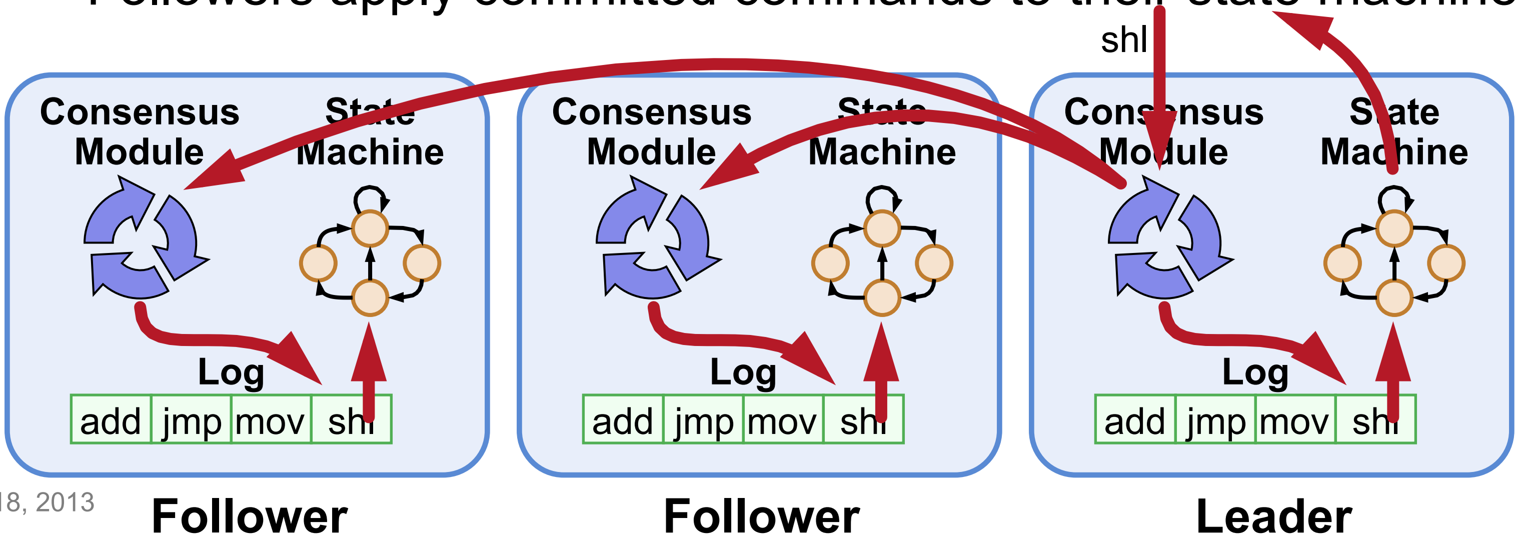
- **Liveness: some candidate must eventually win**
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time

Raft Overview

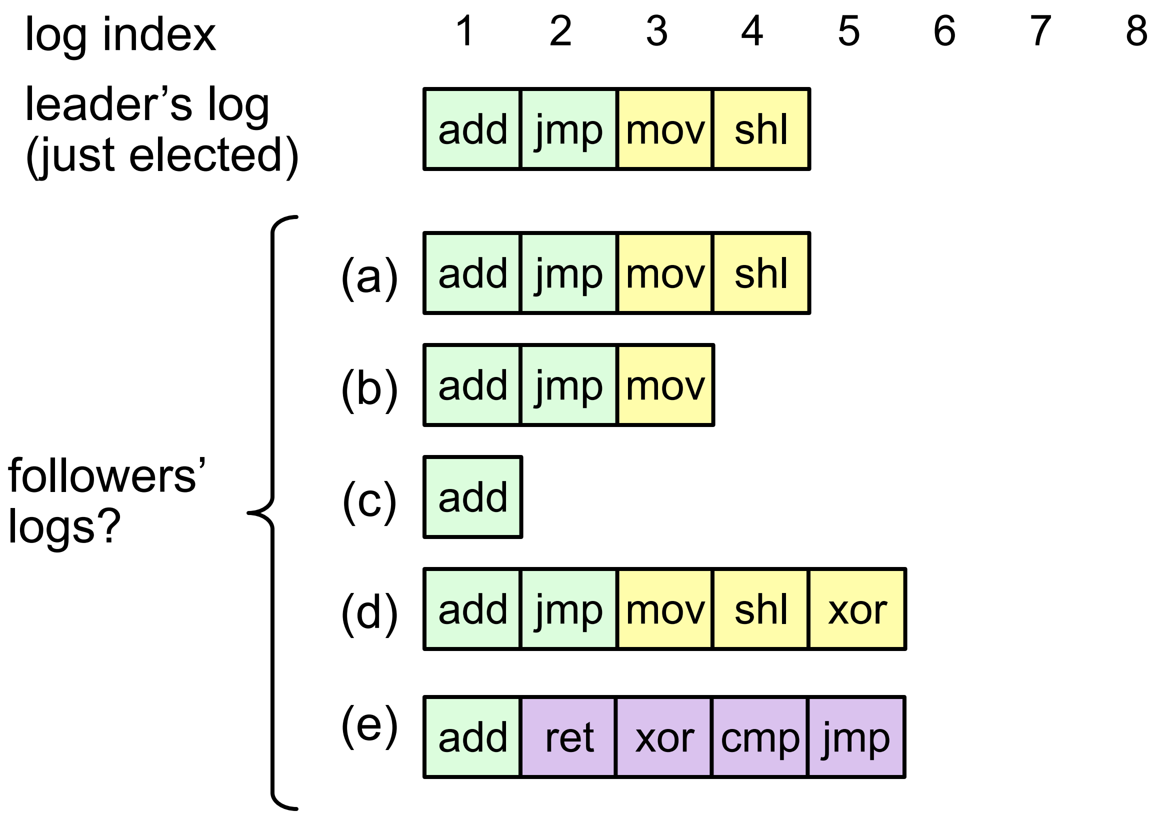
1. **~~Leader election~~:**
   - Selects one of the servers to act as leader
   - Detects crashes, chooses new leader
2. **Normal operation:**
   - Leader accepts client requests
   - Leader blindly overwrites other servers' logs with its own
   - Leader marks entries *committed* when they are safe to apply to state machines
3. **Safety**
   - Rigs leader election to ensure leader's log is "the truth"
   - Defines which entries may be marked committed

Normal Operation

- **Client sends command to leader**
- **Leader appends command to its log**
- **Leader sends AppendEntries RPCs to followers**
- **Once new entry committed:**
  - Leader applies cmd to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers apply committed commands to their state machines

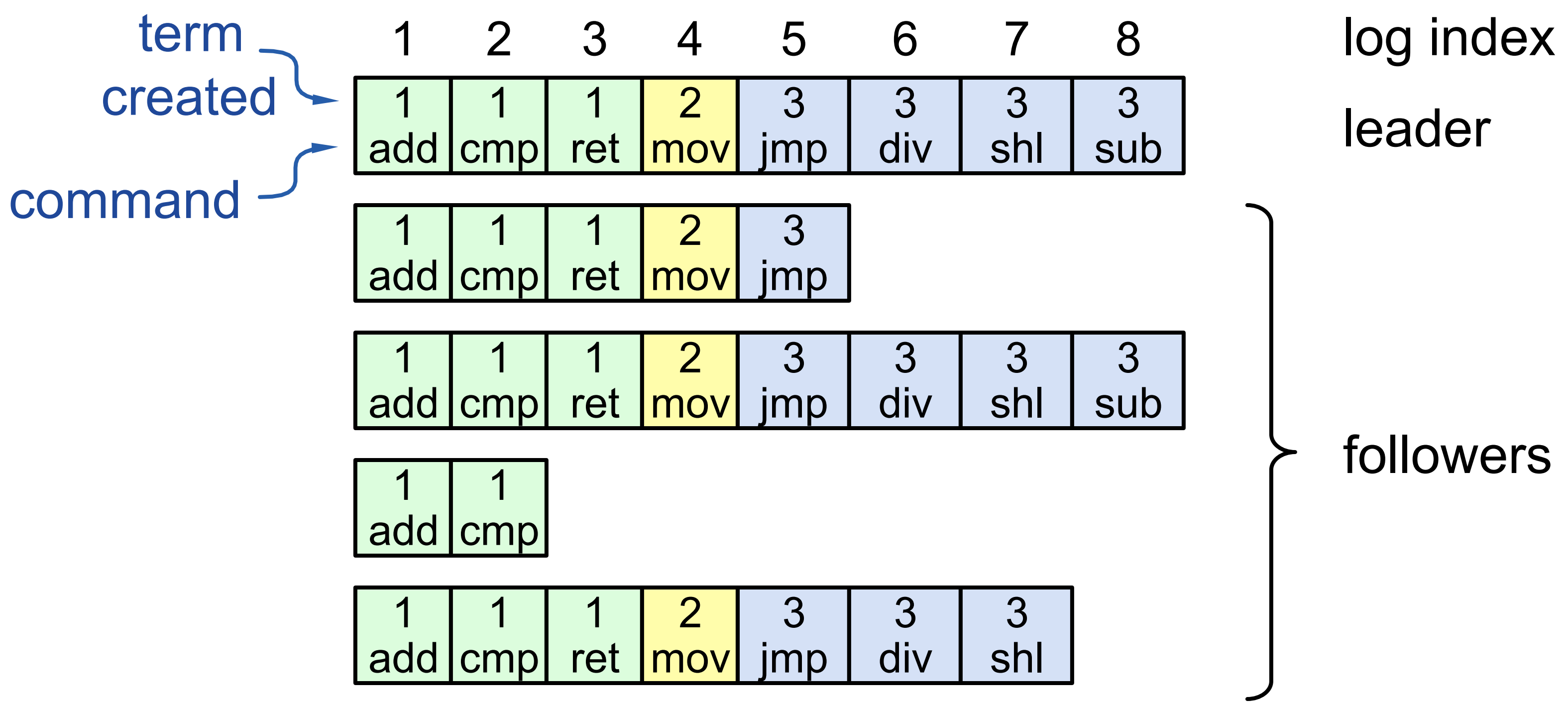


How can follower's logs differ?



What's the easiest way to make the followers' logs match the leader's?

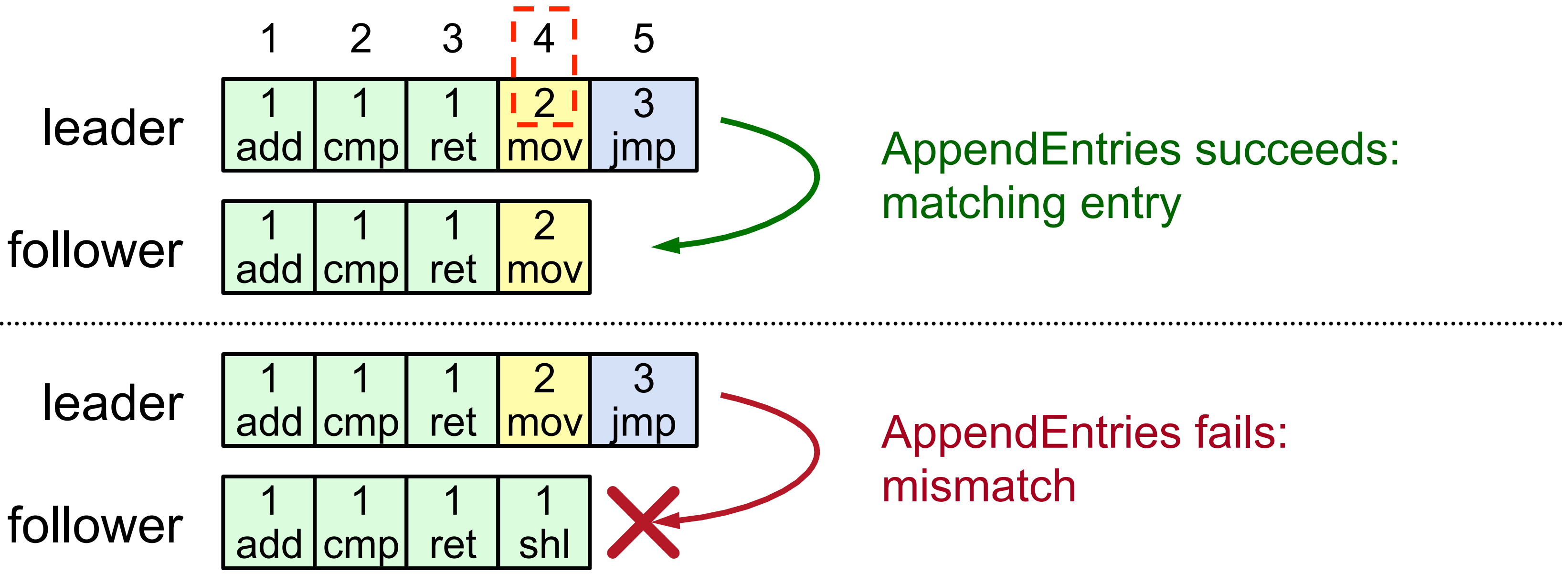
Log Entry Contains Term Created


term created
command
1 2 3 4 5 6 7 8
log index
leader
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl | 3 sub
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl | 3 sub
1 add | 1 cmp
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl
followers

- **If log entries on different servers have same index and term, they store the same command**

AppendEntries Consistency Check

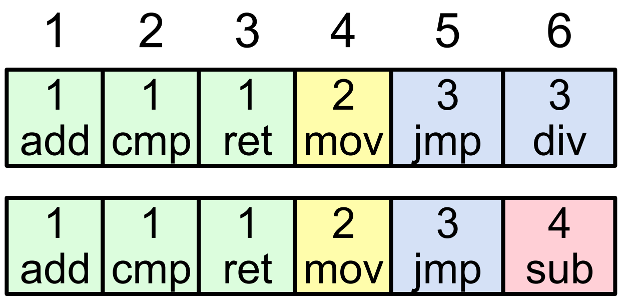
- **Each AppendEntries RPC contains index, term of preceding entry**
- **Follower must contain matching entry; otherwise it rejects request**
- **Implements an induction step, ensures coherency**



Log Consistency

High level of coherency between logs:

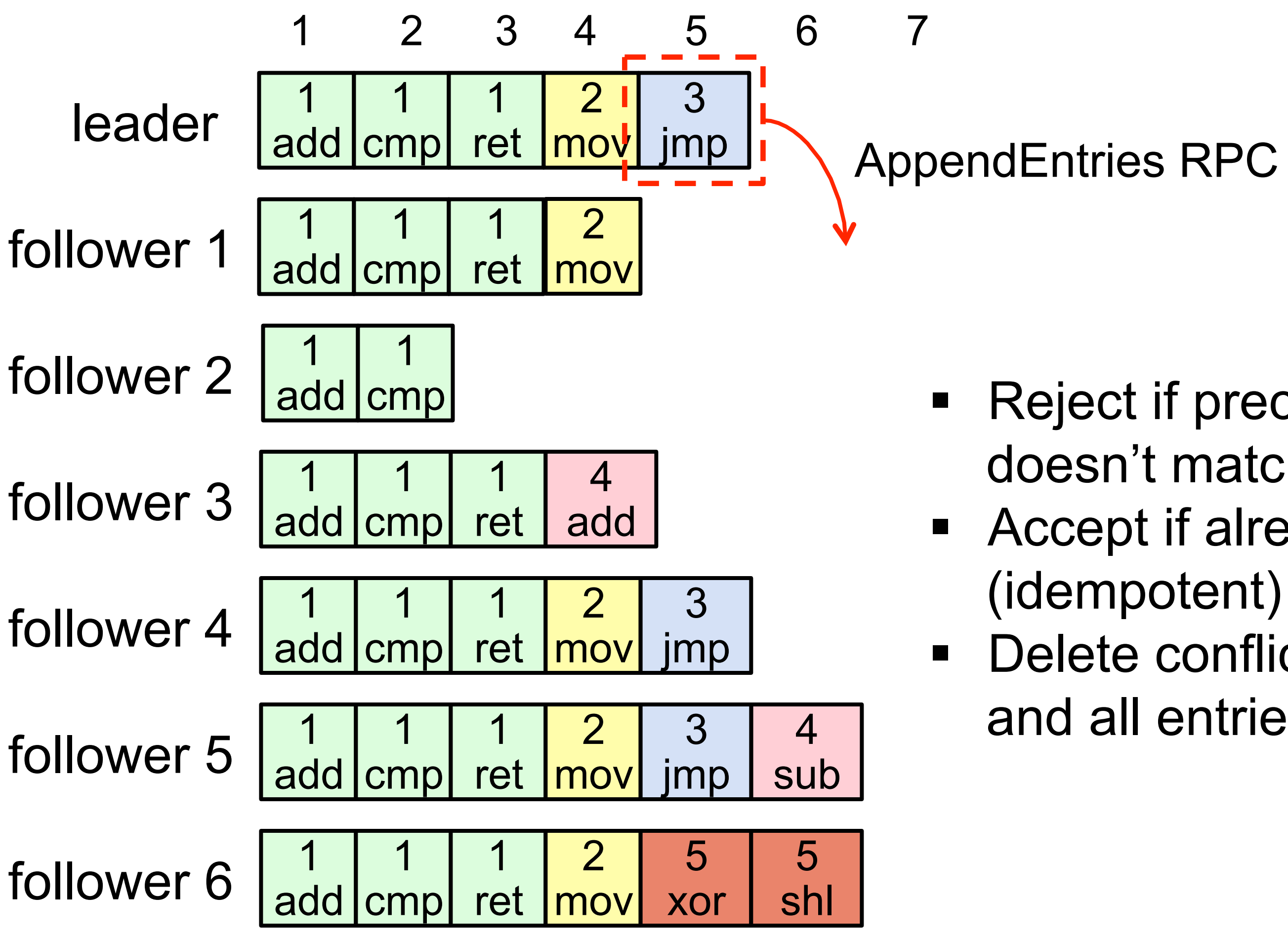
- **If log entries on different servers have same index and term:**
  - They store the same command
  - The logs are identical in all preceding entries



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

AppendEntries Behavior

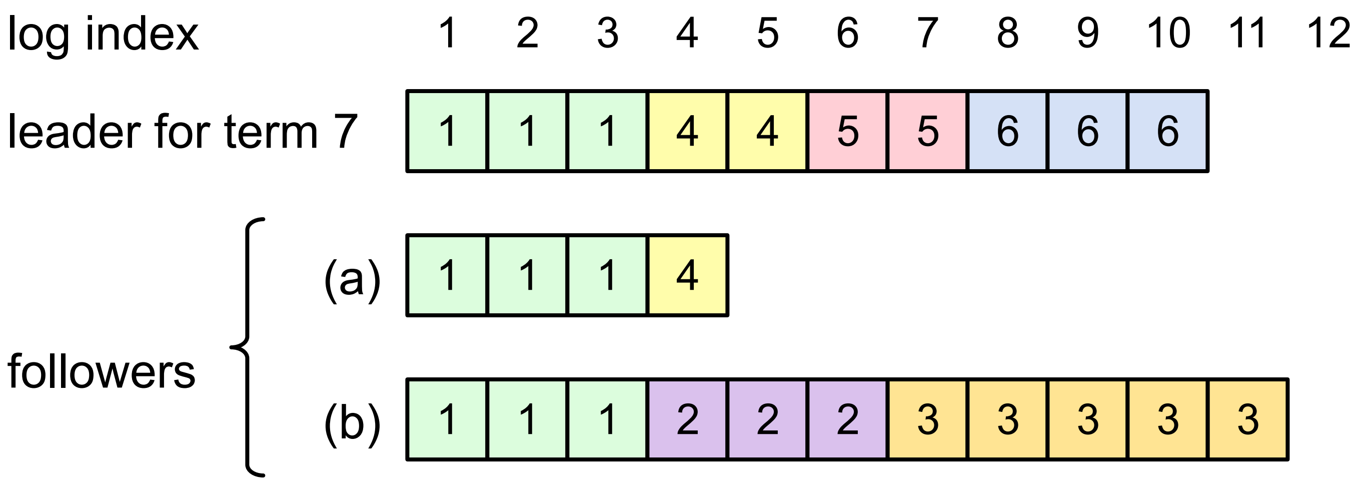
- **What should each follower do upon receiving the AppendEntries RPC?**



- Reject if preceding entry doesn't match
- Accept if already done (idempotent)
- Delete conflicting entries and all entries following

Repairing Follower Logs

- **New leader must make follower logs consistent with its own**
  - Delete extraneous entries
  - Fill in missing entries
- **Needs to send the right entries to make progress**
  - Index too high => rejected
  - Index too low => redundant
  - Index just right => progress
- **How does a leader set its nextIndex?**



Normal Operation Summary

- **Safety: logs can only converge towards the leader's**
  - AppendEntries consistency check guarantees this
  - Next section explains why the leader's log is "the truth"

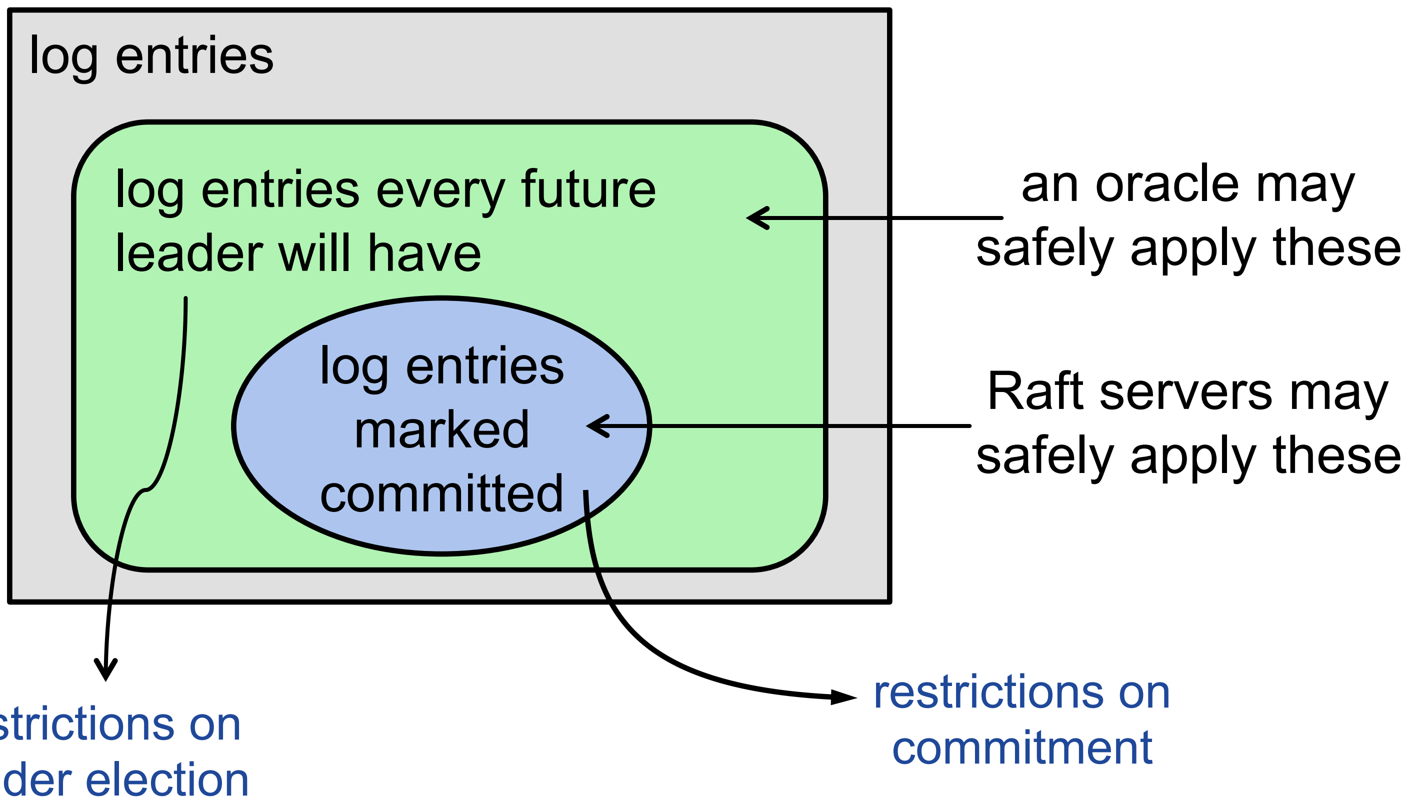
- **Liveness: replication makes progress**
  - Searching for where logs diverge is O(num entries) or O(lg(num entries)) per server
  - Replication takes O(num entries) per server
  - Assuming you can maintain your leadership with heartbeats
    - Heartbeats are just AppendEntries RPCs with no entries

Raft Overview

1. **Leader election:**
   - Selects one of the servers to act as leader
   - Detects crashes, chooses new leader
2. **Normal operation:**
   - Leader accepts client requests
   - Leader blindly overwrites other servers' logs with its own
   - Leader marks entries *committed* when they are safe to apply to state machines
3. **Safety**
   - Rigs leader election to ensure leader's log is "the truth"
   - Defines which entries may be marked committed

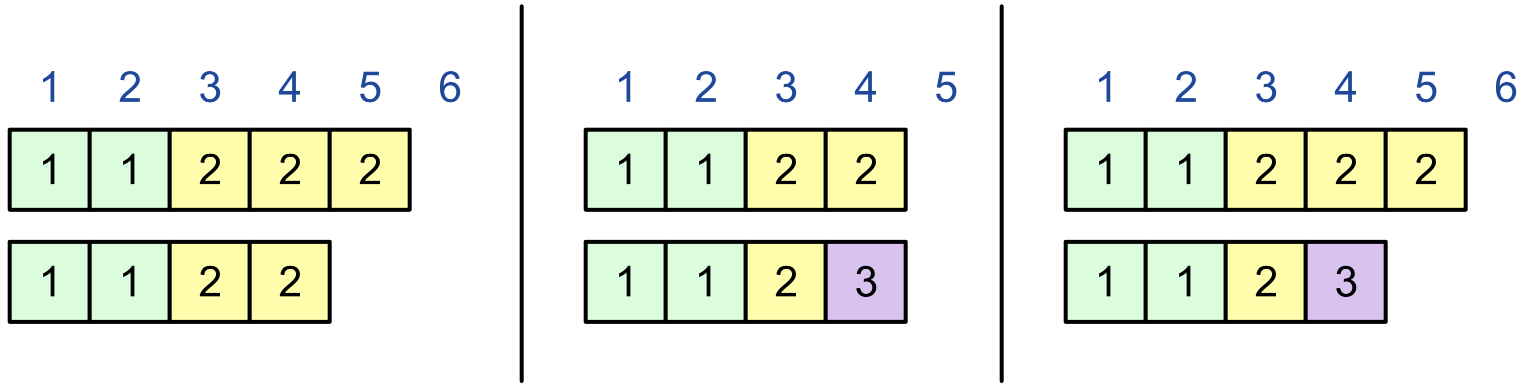
Safety Requirement

Once a log entry has been applied to a state machine, no other state machine may apply a different command at the same log index


log entries
log entries every future leader will have
log entries marked committed
an oracle may safely apply these
Raft servers may safely apply these
restrictions on leader election
restrictions on commitment

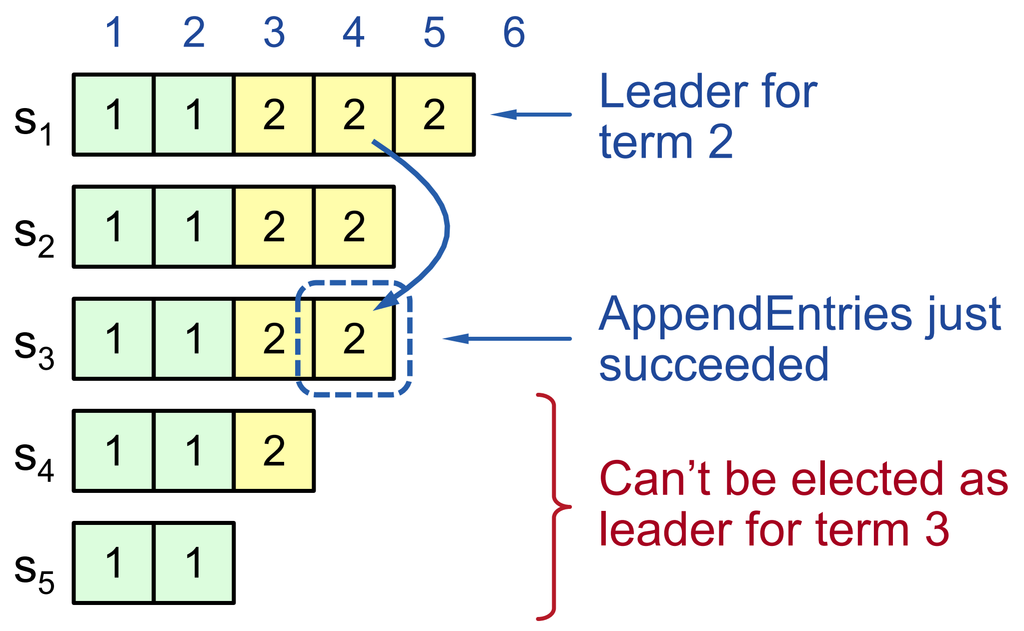
Picking the Best Leader

- **Intuitively, we don't want a server that's been hibernating to become leader**
- **Which log is better?**



- **Voting server denies vote if its log is more up-to-date:**
  - Has higher last term, or same last term but longer log
- **Leader will have most up-to-date log among electing majority**

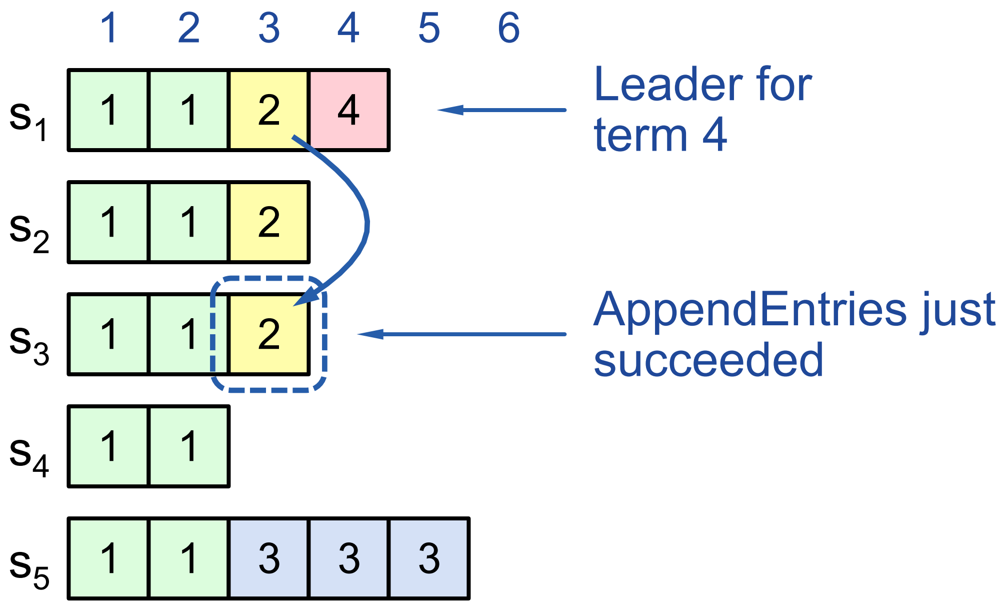
Committing Entry from Current Term



- **Leader for next term must contain entry 4, so it is safe to mark committed and apply**
- **Rule: for entries in current term, need majority**

Committing Entry from Earlier Term

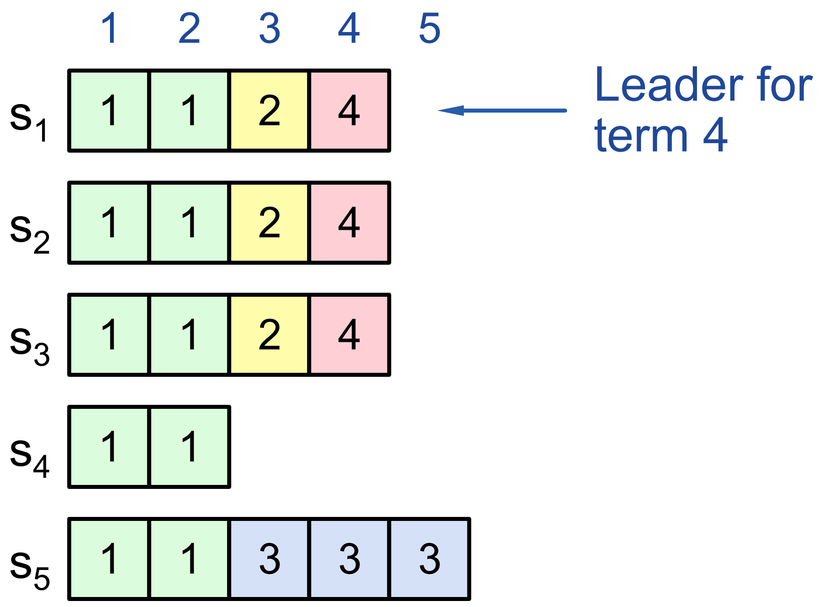
- **Case #2/2: Leader is trying to finish committing entry from an earlier term**



- **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!

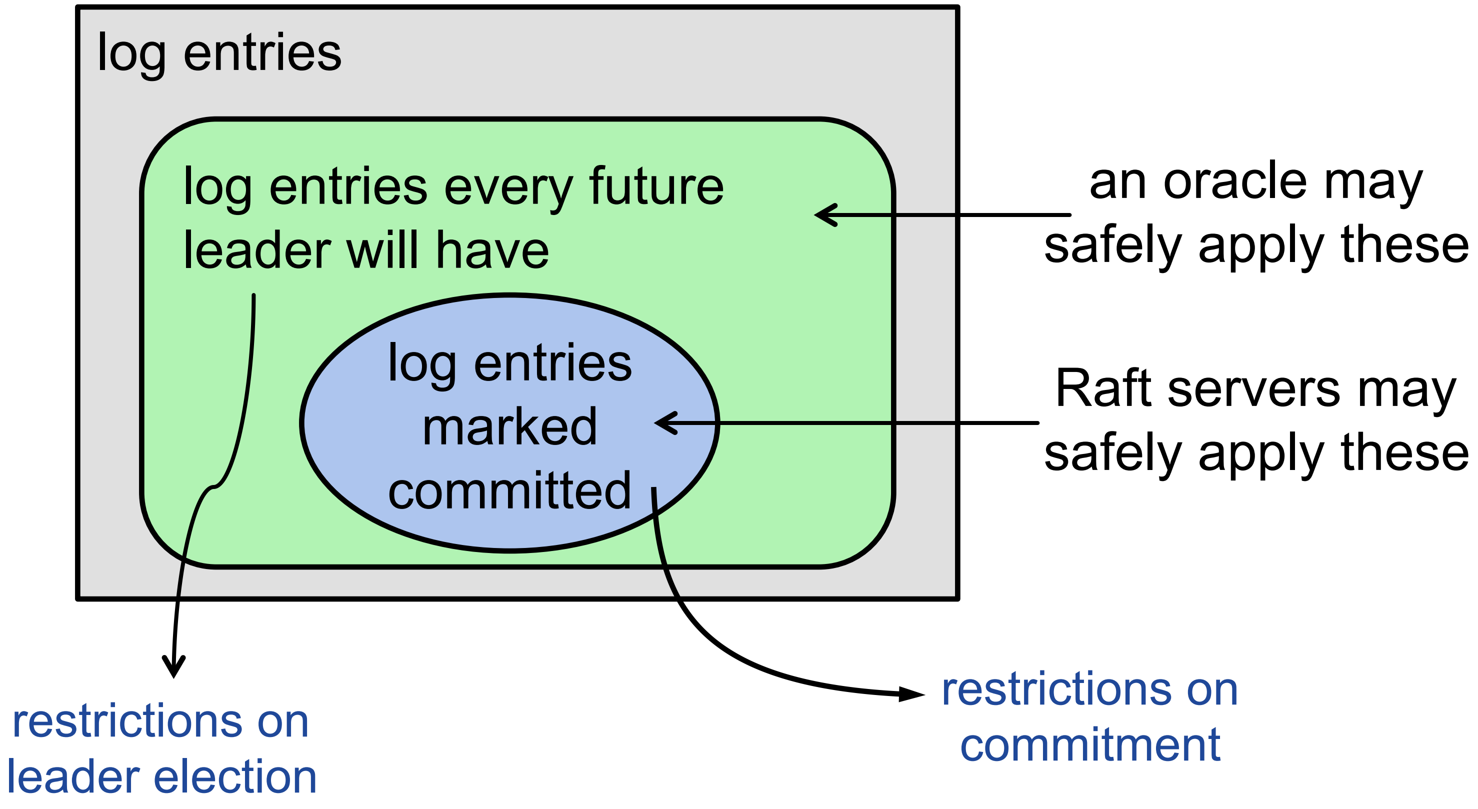
Full Commitment Rule

- **For a leader to decide an entry is committed:**
  - Must be from current term and stored on a majority of servers
  - Commitment of entries from prior terms simply delayed
- **Once entry 4 committed:**
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Safety Summary

Once a log entry has been applied to a state machine, no other state machine may apply a different command at the same log index


log entries
log entries every future leader will have
log entries marked committed
an oracle may safely apply these
Raft servers may safely apply these
restrictions on leader election
restrictions on commitment

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 Summary

1. **Leader election**
2. **Normal operation**
3. **Safety and consistency**

- **Questions?**
- **http://ramcloud.stanford.edu/raft.pdf**
  - Summary sheet, more details and explanations
  - Pointers to video lectures, formal specification, C++ implementation
- **ongaro@cs.stanford.edu**

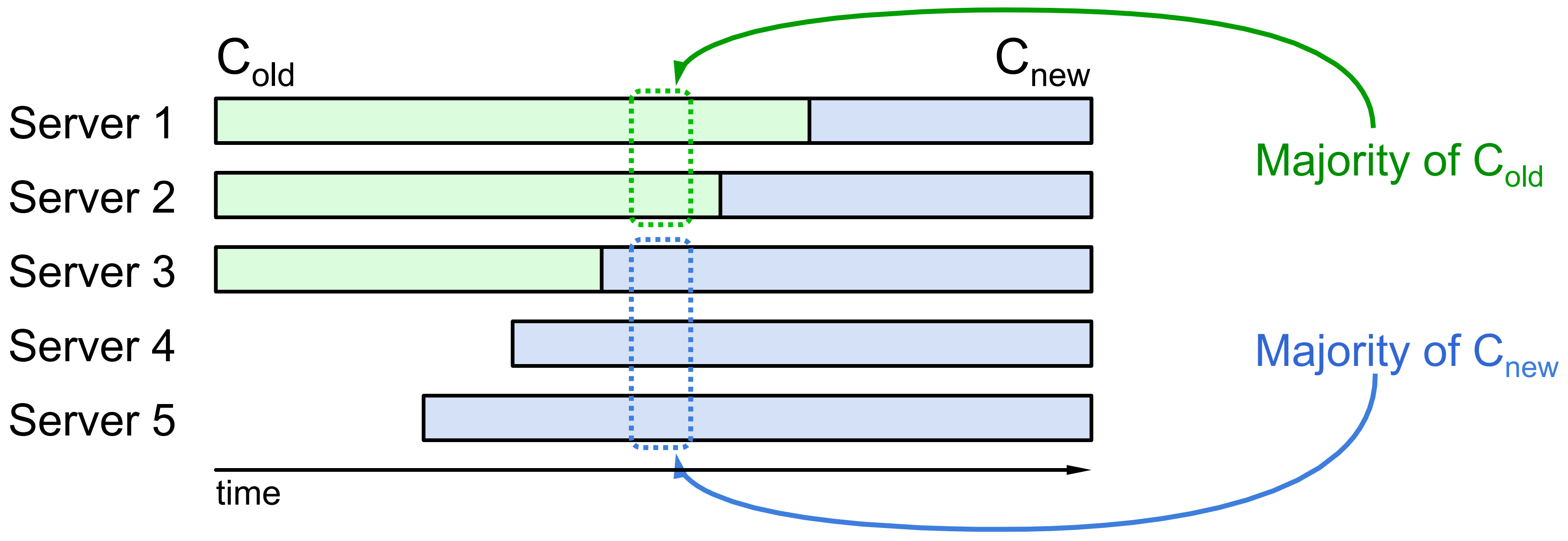
Extra slides

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

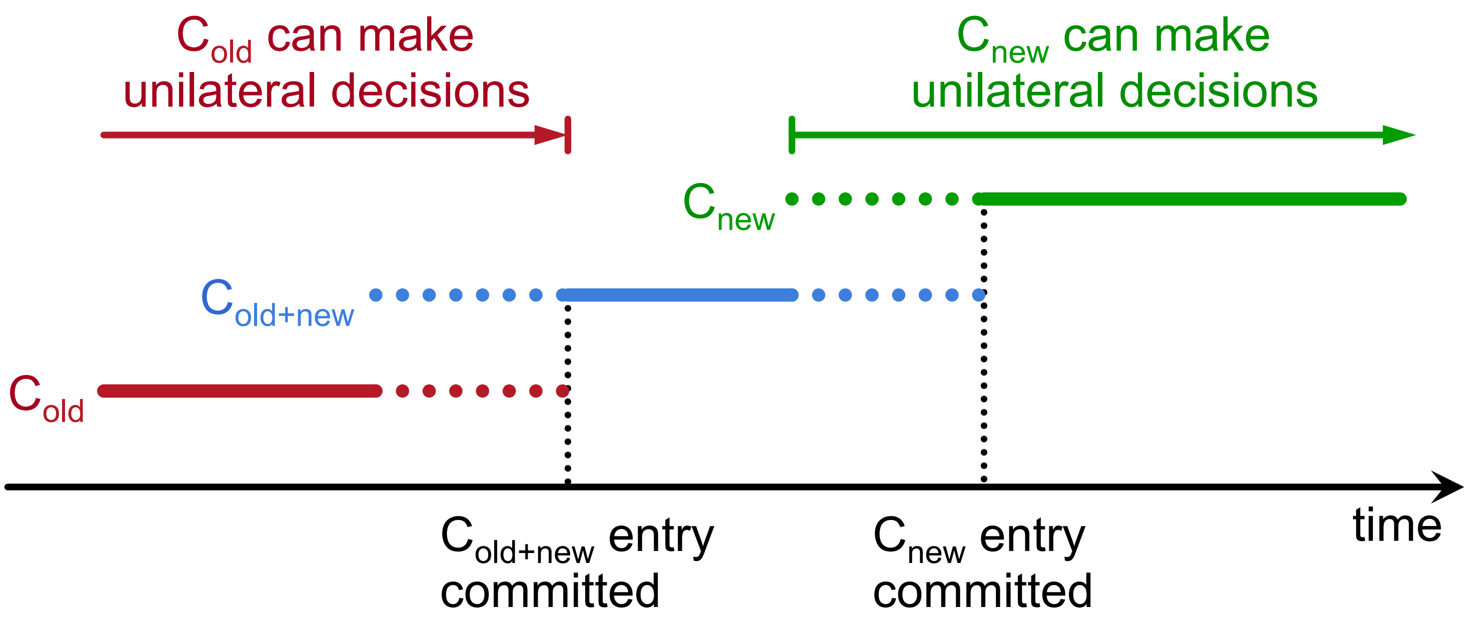
Configuration Changes, cont'd

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



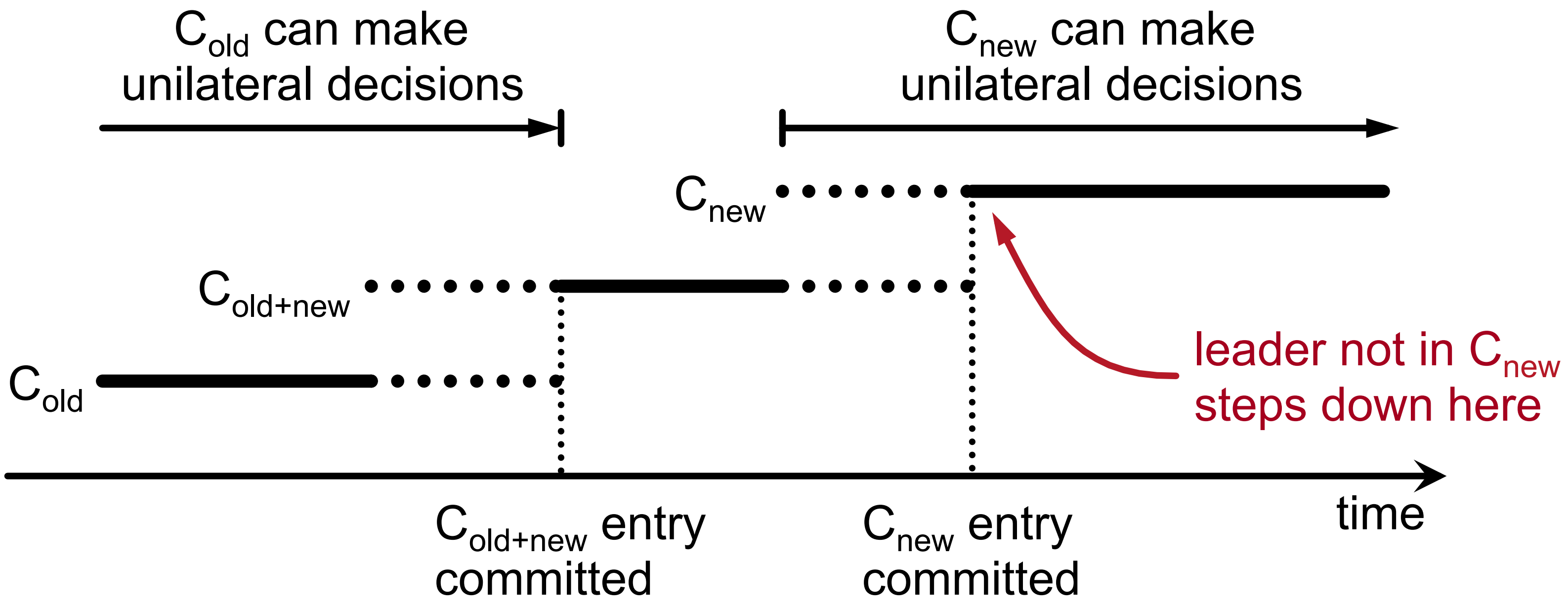
Joint Consensus

- **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


C_{old} can make unilateral decisions
C_{new} can make unilateral decisions
C_{new}
$C_{old+new}$
C_{old}
$C_{old+new}$ entry committed
C_{new} entry committed
time

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.



C_{old} can make unilateral decisions
C_{new} can make unilateral decisions
C_{new}
$C_{old+new}$
C_{old}
leader not in C_{new} steps down here
$C_{old+new}$ entry committed
C_{new} entry committed
time

Safety Property

Once a log entry has been applied to a state machine, no other state machine may apply a different command at the same log index

- **Only entries marked committed are applied**
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
  - If an entry is present in the logs of all future leaders, it is safe to apply to state machines (clients only talk to leaders)

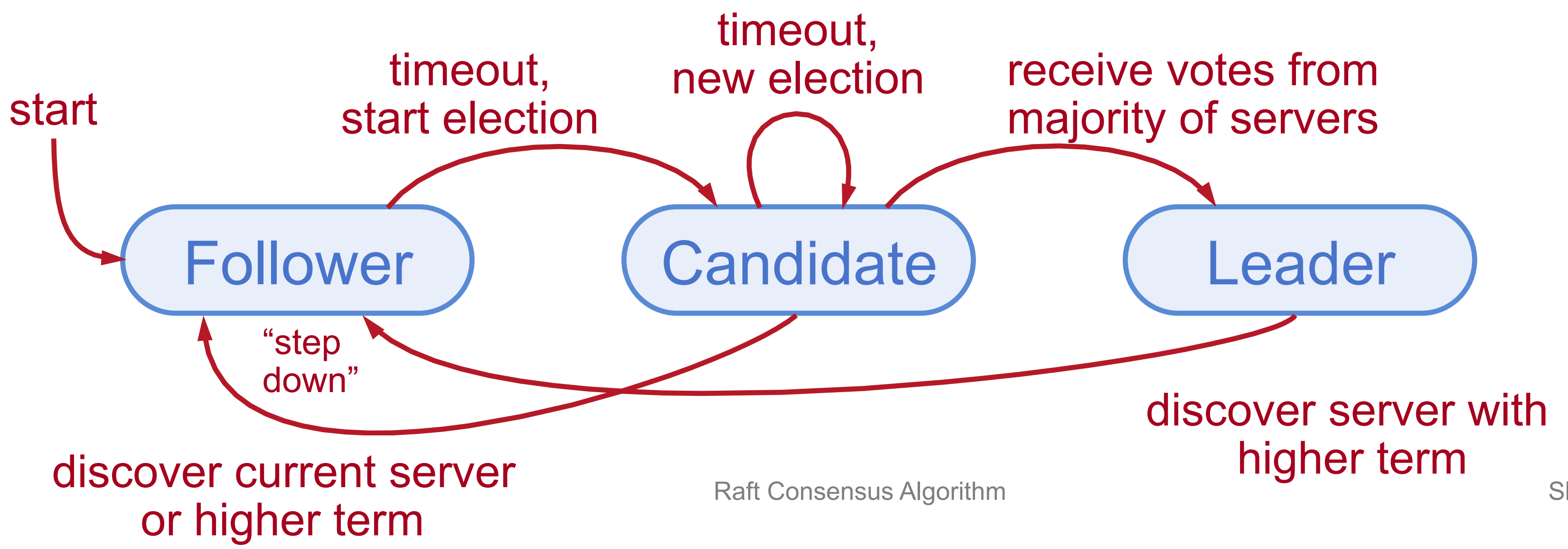
Marked Committed => Present in future leaders' logs => Safe to apply

Restrictions on commitment

Restrictions on leader election

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**


start
timeout, start election
timeout, new election
receive votes from majority of servers
Follower
Candidate
Leader
"step down"
discover current server or higher term
discover server with higher term

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

Client Protocol

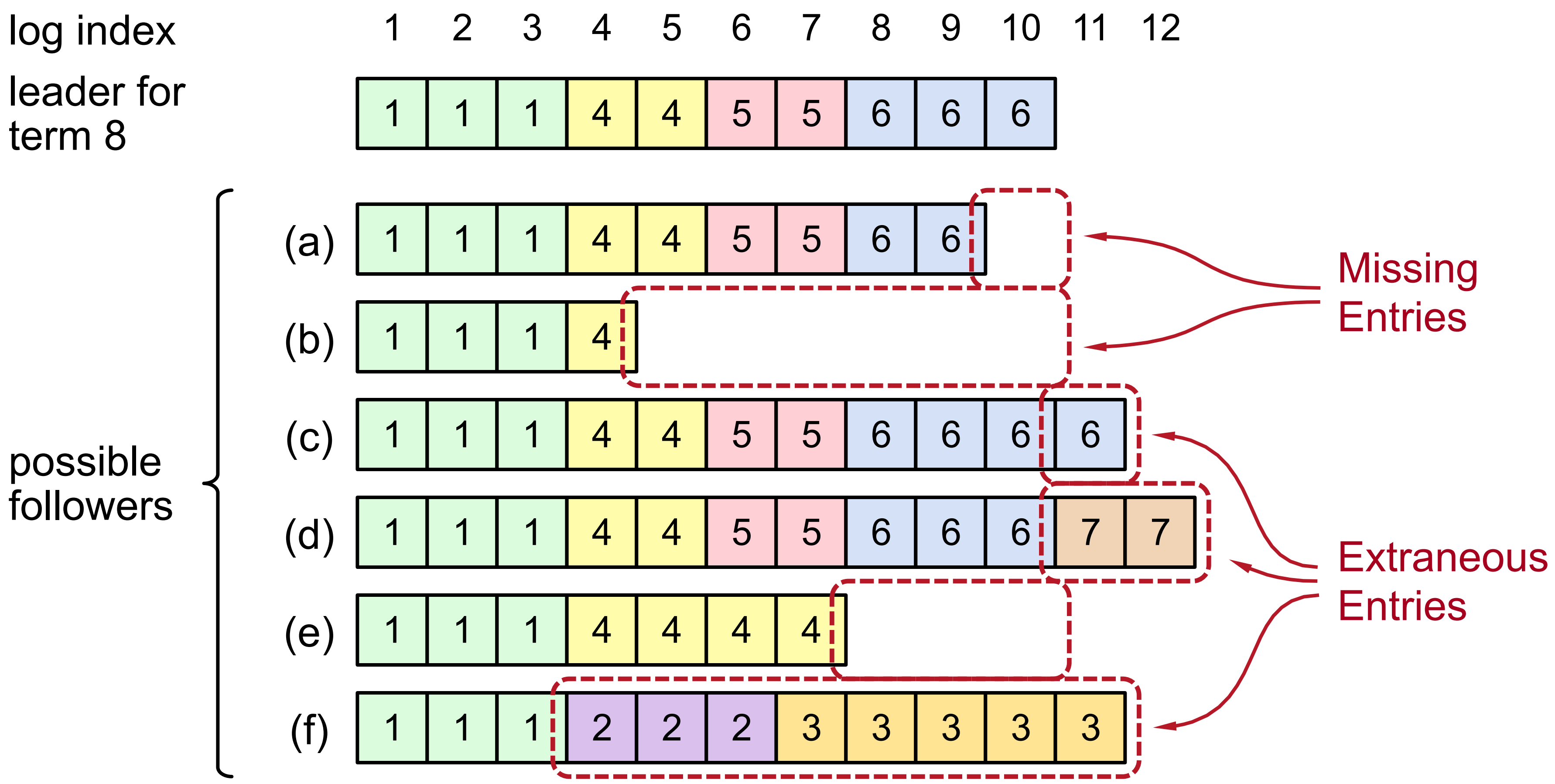
- **Send commands to leader**
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
- **Leader does not respond until command has been logged, committed, and executed by leader's state machine**
- **If request times out (e.g., leader crash):**
  - Client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

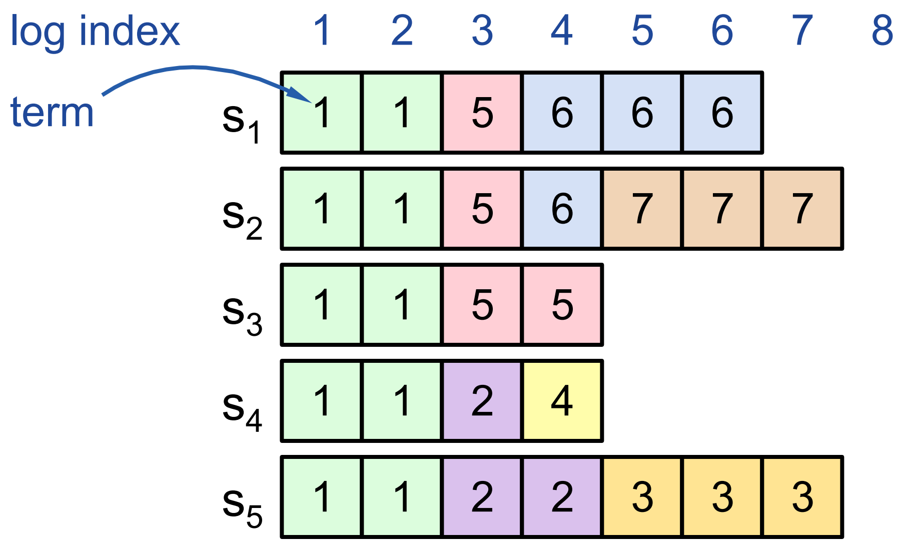
Log Inconsistencies

Leader changes can result in log inconsistencies:



Leader Changes

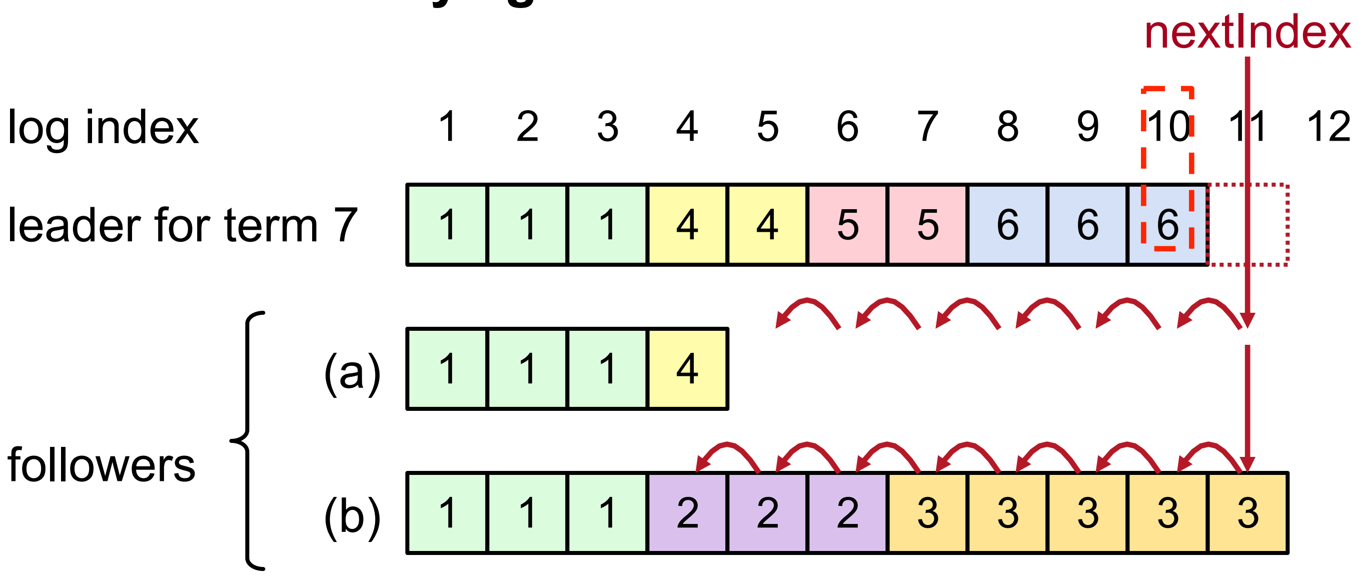
- **At beginning of new leader's term:**
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous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



	1	2	3	4	5	6	7	8
s_1	1	1	5	6	6	6		
s_2	1	1	5	6	7	7	7	
s_3	1	1	5	5				
s_4	1	1	2	4				
s_5	1	1	2	2	3	3	3	

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:**

				nextIndex ↓							
log index	1	2	3	4	5	6	7	8	9	10	11
leader for term 7	1	1	1	4	4	5	5	6	6	6	
follower (before)	1	1	1	2	2	2	3	3	3	3	3
follower (after)	1	1	1	4							