Consensus: Paxos and RAFT

COS 418: Distributed Systems
Lecture 13

Wyatt Lloyd

RAFT slides based on those from Diego Ongaro and John Ousterhout

Consensus Used in Systems

Group of servers want to:

- Make sure all servers in group receive the same updates in the same order as each other
- Maintain own lists (views) on who is a current member of the group, and update lists when somebody leaves/fails
- Elect a leader in group, and inform everybody
- Ensure mutually exclusive (one process at a time only) access to a critical resource like a file

Single-shot Consensus

- Figure out how to reach consensus for 1 decision

Paxos Guarantees

- Safety (bad things never happen)
    - **Agreement:** All processes that decide do so on the same value
    - **Validity:** Value decided must have proposed by some process
- Liveness (good things eventually happen)
    - **Termination:** All non-faulty processes eventually decide on a value

Paxos's Safety and Liveness

- Paxos is always safe
- Paxos is very often live (but not always, more later)

Roles of a Process in Paxos

- Three conceptual roles
  - Proposers propose values
  - Acceptors accept values, where value is chosen if majority accept
  - Learners learn the outcome (chosen value)

- In reality, a process can play any/all roles

Strawmen

- 3 proposers, 1 acceptor
  - Acceptor accepts first value received
  - No liveness with single failure

- 3 proposers, 3 acceptors
  - Accept first value received, learners choose common value known by majority
  - But no such majority is guaranteed

Paxos

- Each acceptor accepts multiple proposals
  - Hopefully one of multiple accepted proposals will have a majority vote (and we determine that)
  - If not, rinse and repeat (more on this)

- How do we select among multiple proposals?
  - Ordering: proposal is tuple (proposal #, value) = (n, v)
  - Proposal # strictly increasing, globally unique
  - Globally unique?
    - Trick: set low-order bits to proposer's ID

Paxos Protocol Overview

- Proposers:
  1. Choose a proposal number n
  2. Ask acceptors if any accepted proposals with $n_a < n$
  3. If existing proposal v_a returned, propose same value (n, v_a)
  4. Otherwise, propose own value (n, v)

  Note altruism: goal is to reach consensus, not “win”

- Accepters try to accept value with highest proposal n
- Learners are passive and wait for the outcome

Paxos Phase 1

- Proposer:
  - – Choose proposal n, send <prepare, n> to acceptors

- Acceptors:
  - If $n > n_h$
    - $n_h = n$ ← promise not to accept any new proposals n' < n
    - If no prior proposal accepted
      - Reply < promise, n, Ø >
    - Else
      - Reply < promise, n, (n_a, v_a) >
  - Else
    - Reply < prepare-failed >

Paxos Phase 2

- Proposer:
  - If receive promise from majority of acceptors,
    - Determine v_a returned with highest n_a, if exists
    - Send <accept, (n, v_a || v)> to acceptors

- Acceptors:
  - Upon receiving (n, v), if $n \geq n_h$,
    - Accept proposal and notify learner(s)

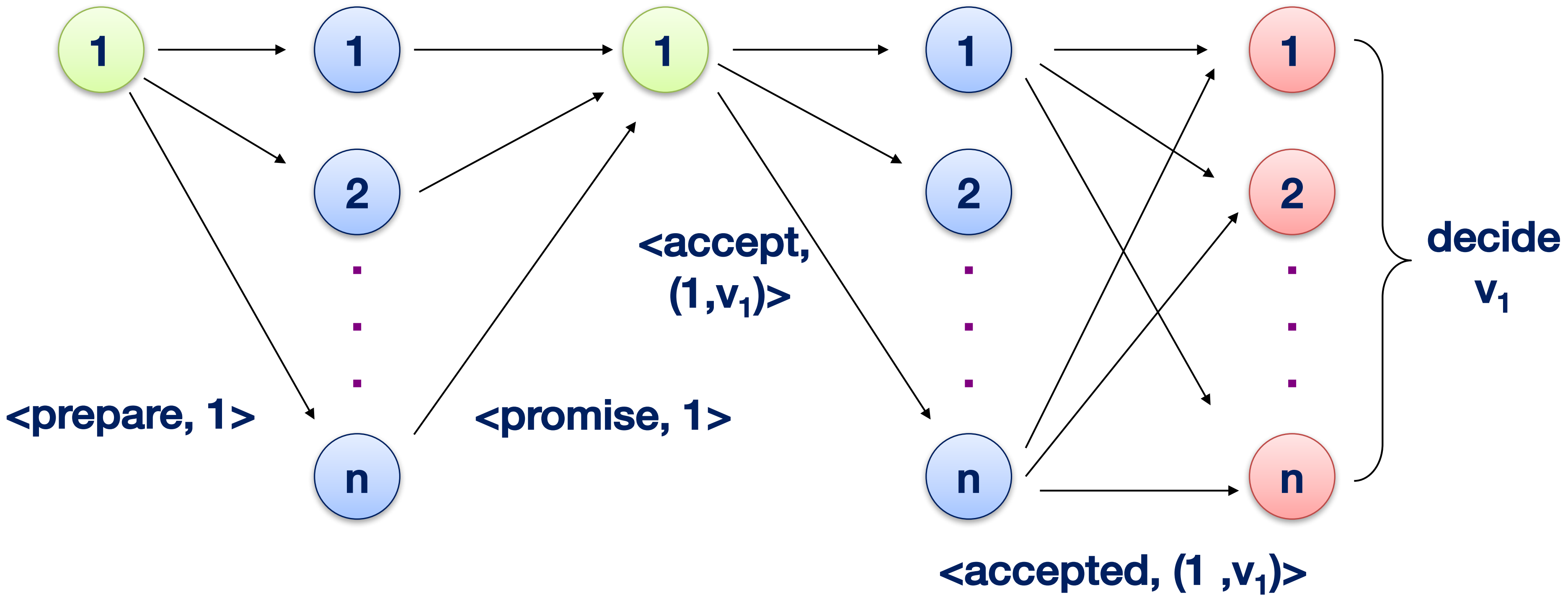
$$n_a = n_h = n$$
$$v_a = v$$

Paxos Phase 3

- Learners need to know which value chosen

- Each acceptor notifies all learners
  - Simplest approach, but many messages

Paxos: Well-behaved Run



<prepare, 1>

<promise, 1>

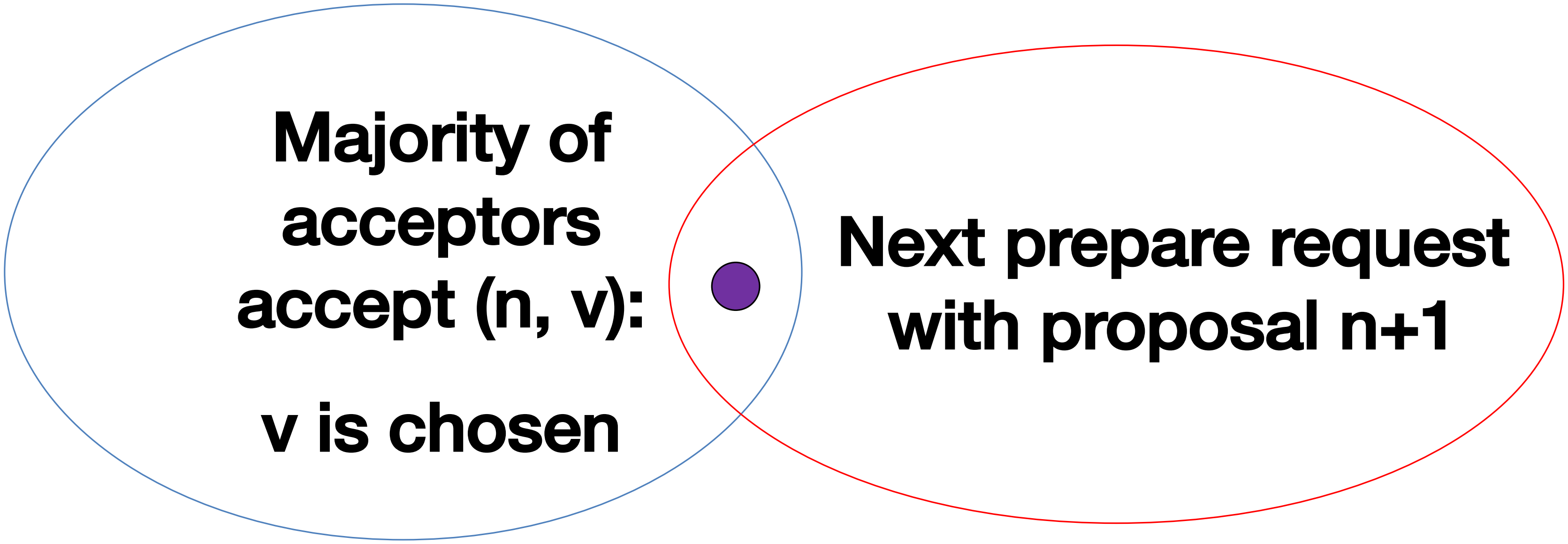
<accept, $(1,v_1)$>

<accepted, $(1 ,v_1)$>

decide v_1

Paxos is Safe

- Intuition: if proposal with value v chosen, then every higher-numbered proposal issued by any proposer has value v.

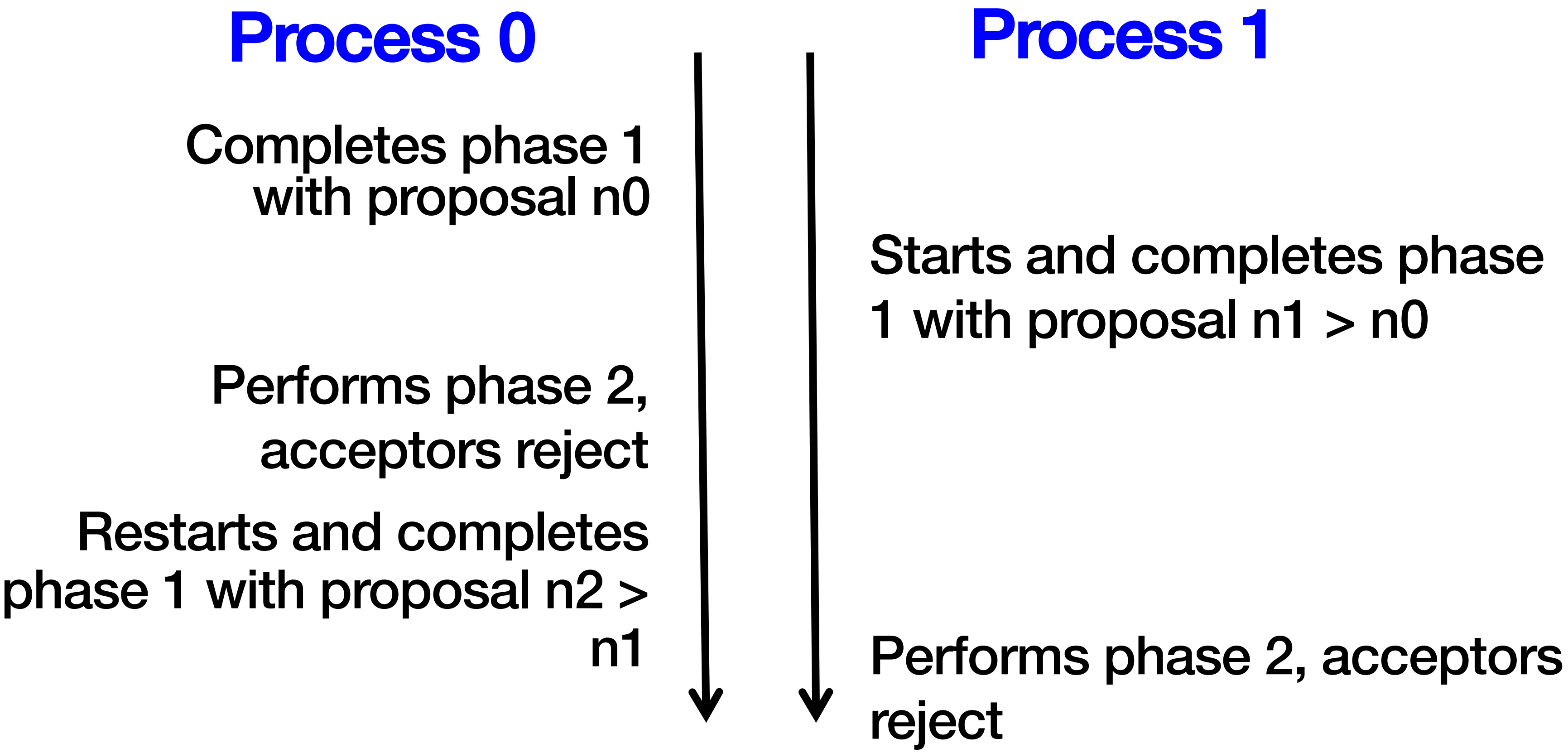


Majority of acceptors accept (n, v):

v is chosen

Next prepare request with proposal n+1

Often, but not always, live



Process 0

Process 1

Completes phase 1 with proposal n0

Starts and completes phase 1 with proposal n1 > n0

Performs phase 2, acceptors reject

Restarts and completes phase 1 with proposal n2 > n1

Performs phase 2, acceptors reject

… can go on indefinitely …

Paxos Summary

- Described for a single round of consensus
- Proposer, Acceptors, Learners
  - Often implemented with nodes playing all roles

- Always safe: Quorum intersection
- Very often live

- Acceptors accept multiple values
  - But only one value is ultimately chosen
- Once a value is accepted by a majority it is chosen

Flavors of Paxos

- Terminology is a mess
- Paxos loosely and confusingly defined…

- We’ll stick with
  - Basic Paxos
  - Multi-Paxos

Flavors of Paxos: Basic Paxos

- Run the full protocol each time
  - e.g., for each slot in the command log

- Takes 2 rounds until a value is chosen

Flavors of Paxos: Multi-Paxos

- Elect a leader and have them run 2nd phase directly
  - e.g., for each slot in the command log
  - Leader election uses Basic Paxos

- Takes 1 round until a value is chosen
  - Faster than Basic Paxos

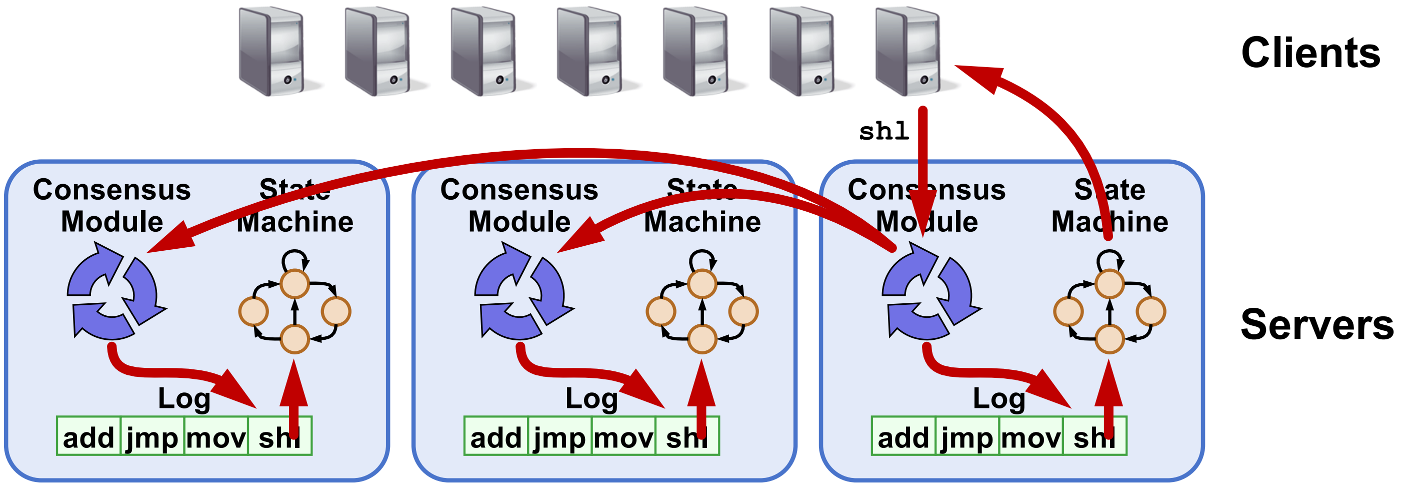
- Used extensively in practice!
  - RAFT is similar to Multi Paxos

RAFT: A CONSENSUS ALGORITHM FOR REPLICATED LOGS

Diego Ongaro and John Ousterhout

Stanford University

Goal: Replicated Log



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

Raft Overview

1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration

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

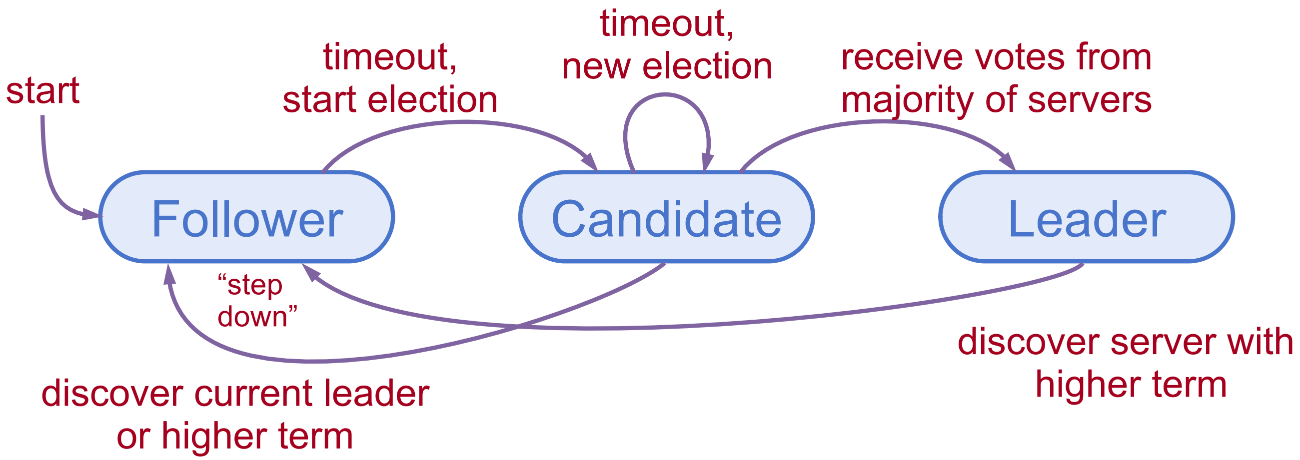
Follower

Candidate

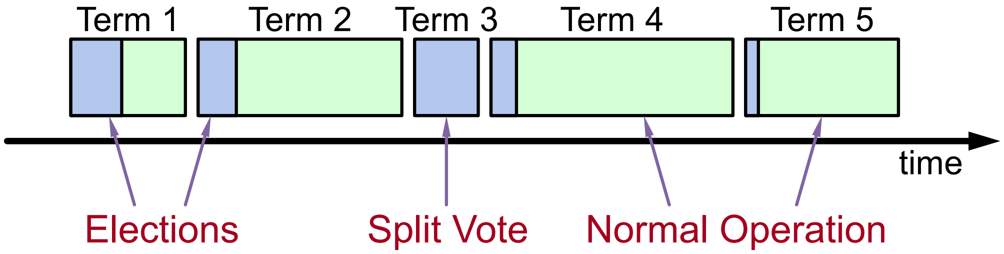
Leader

Liveness Validation

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



Terms (aka epochs)



- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains current term value
- Key role of terms: identify obsolete information

Elections

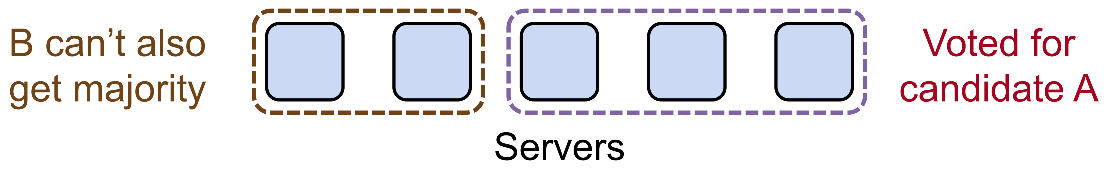
- Start election:
  - Increment current term, change to candidate state, vote for self

- Send RequestVote 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

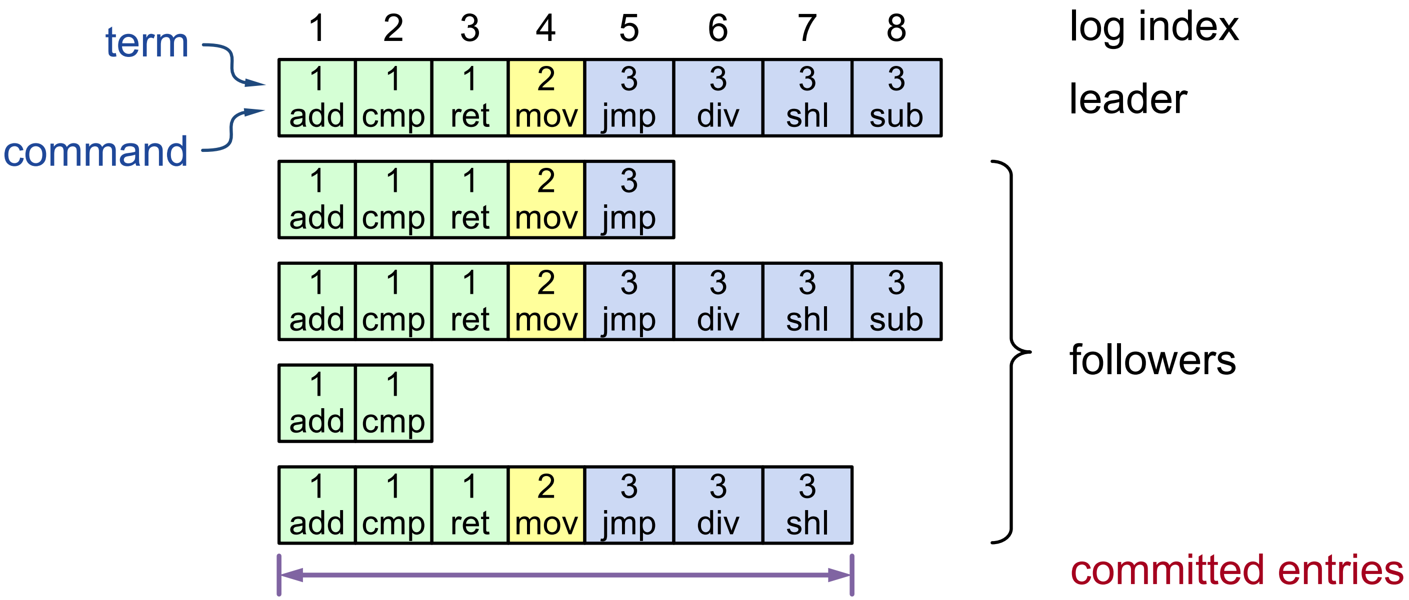
Elections

- 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 choose election timeouts randomly in [T, 2T]
  - One usually initiates and wins election before others start
  - Works well if T >> network RTT

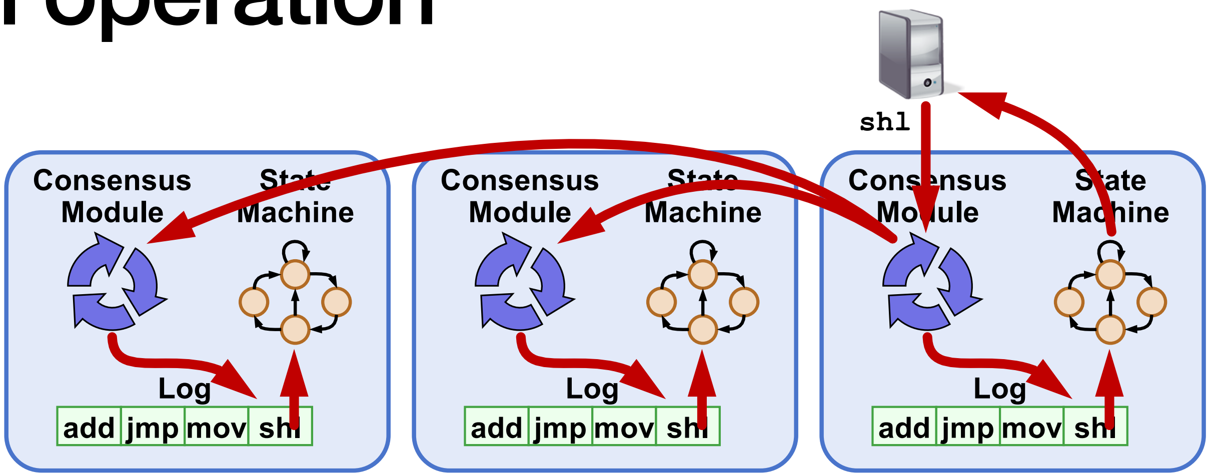
Log Structure



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

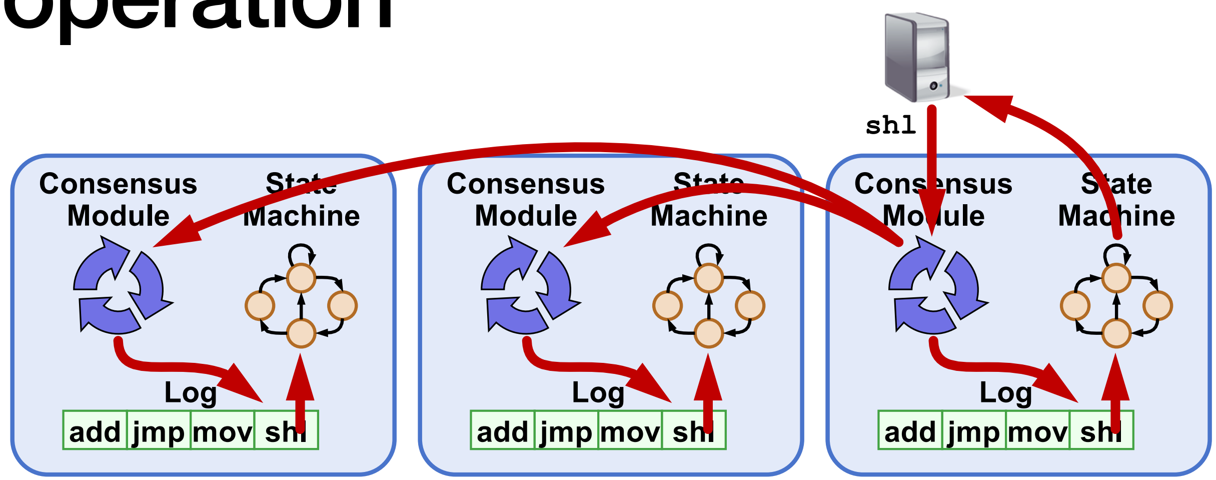
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

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, sends result to client
  - Leader piggybacks commitment to followers in later AppendEntries
  - Followers pass committed commands to their state machines

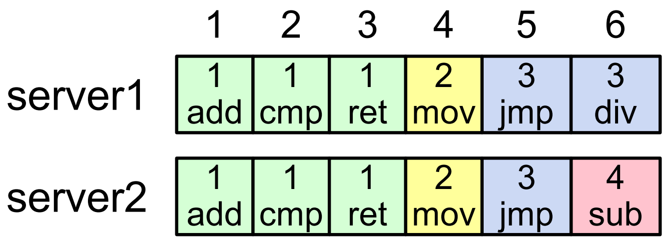
Normal operation



- Crashed / slow followers?
  - Leader retries RPCs until they succeed

- Performance is “optimal” in common case:
  - One successful RPC to any majority of servers

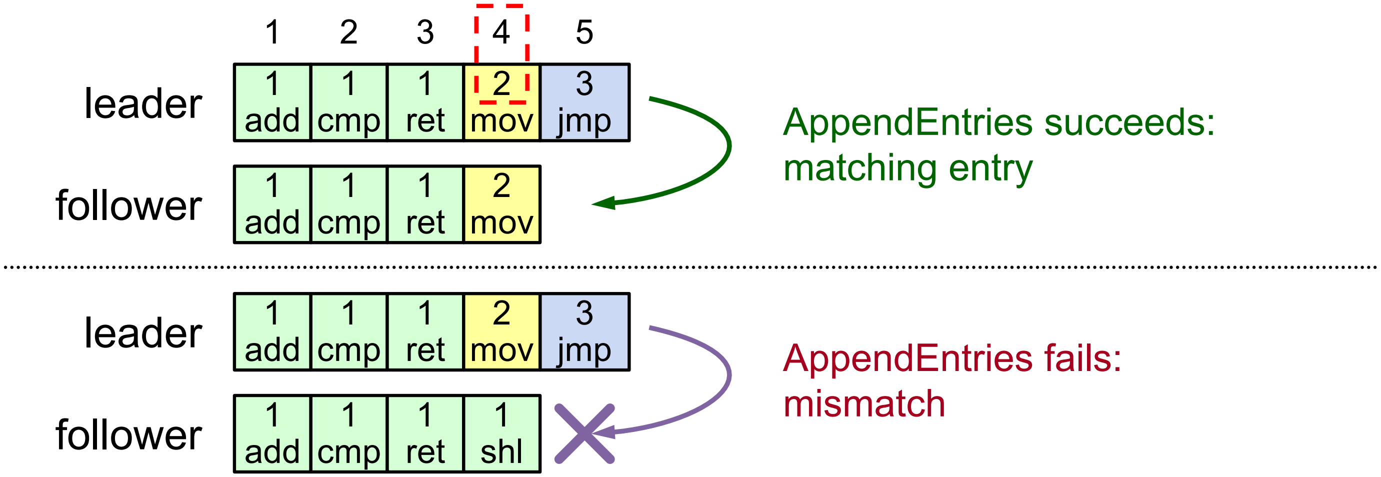
Log Operation: Highly Coherent



	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 server have same index and term:
  - Store the same command
  - Logs are identical in all preceding entries
- If given entry is committed, all preceding also committed

Log Operation: Consistency Check



AppendEntries succeeds:
matching entry

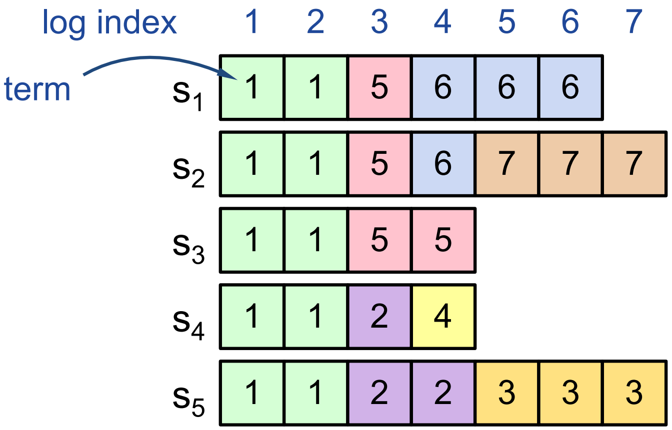
AppendEntries fails:
mismatch

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

Leader Changes

- New leader’s log is truth, no special steps, start 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



log index	1	2	3	4	5	6	7
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

Safety Requirement

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

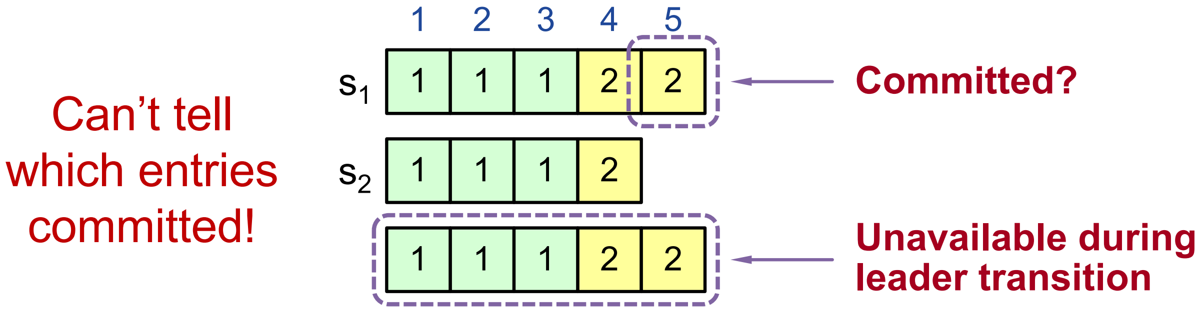
- **Raft safety property:** If leader has decided log entry is committed, entry will be present in logs of all future leaders
- **Why does this guarantee higher-level goal?**
  1. Leaders never overwrite entries in their logs
  2. Only entries in leader's log can be committed
  3. Entries must be committed before applying to state machine

Committed → Present in future leaders' logs

Restrictions on commitment

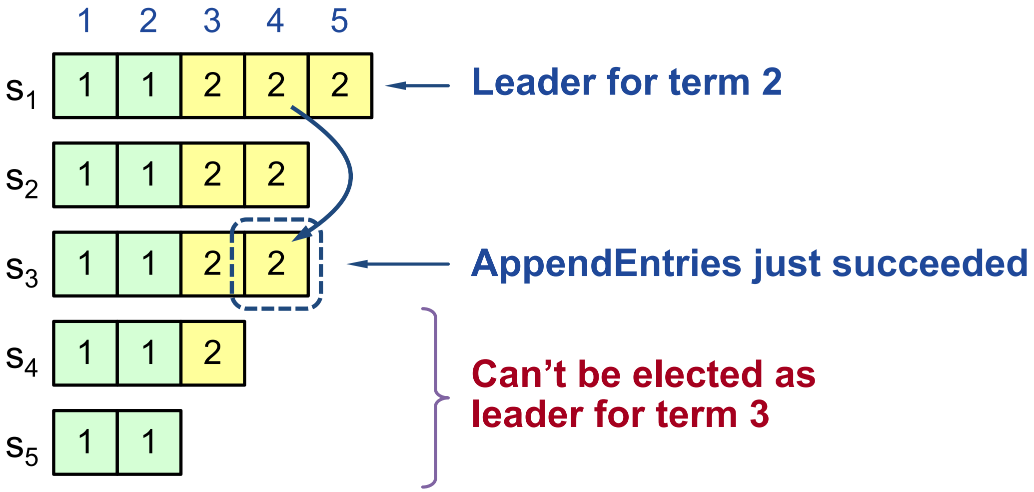
Restrictions on leader election

Picking the Best Leader



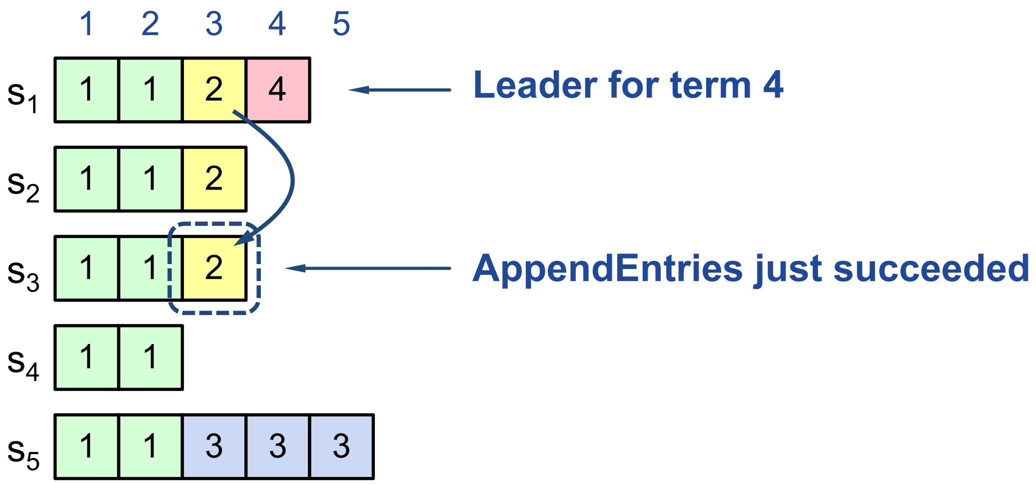
- Elect candidate most likely to contain all committed entries
  - In RequestVote, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete": (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among electing majority

Committing Entry from Current Term



- Case #1: Leader decides entry in current term is committed
- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

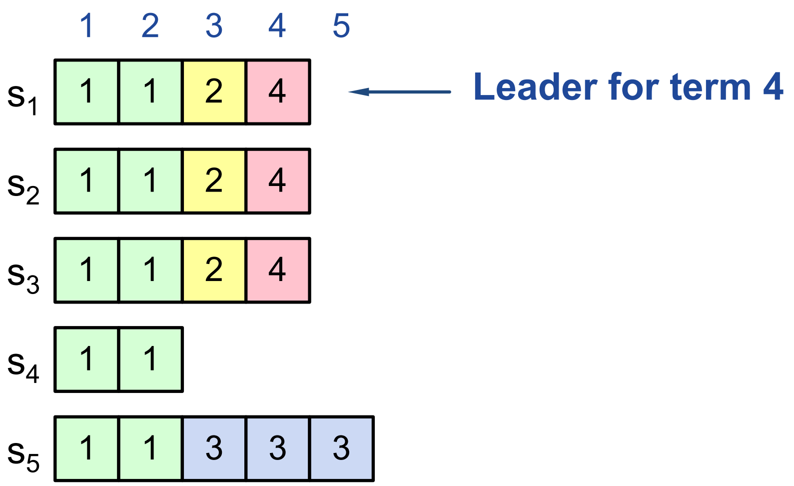


Leader for term 4

AppendEntries just succeeded

- Case #2: Leader trying to finish committing entry from earlier
- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5 (how?)
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3

New Commitment Rules

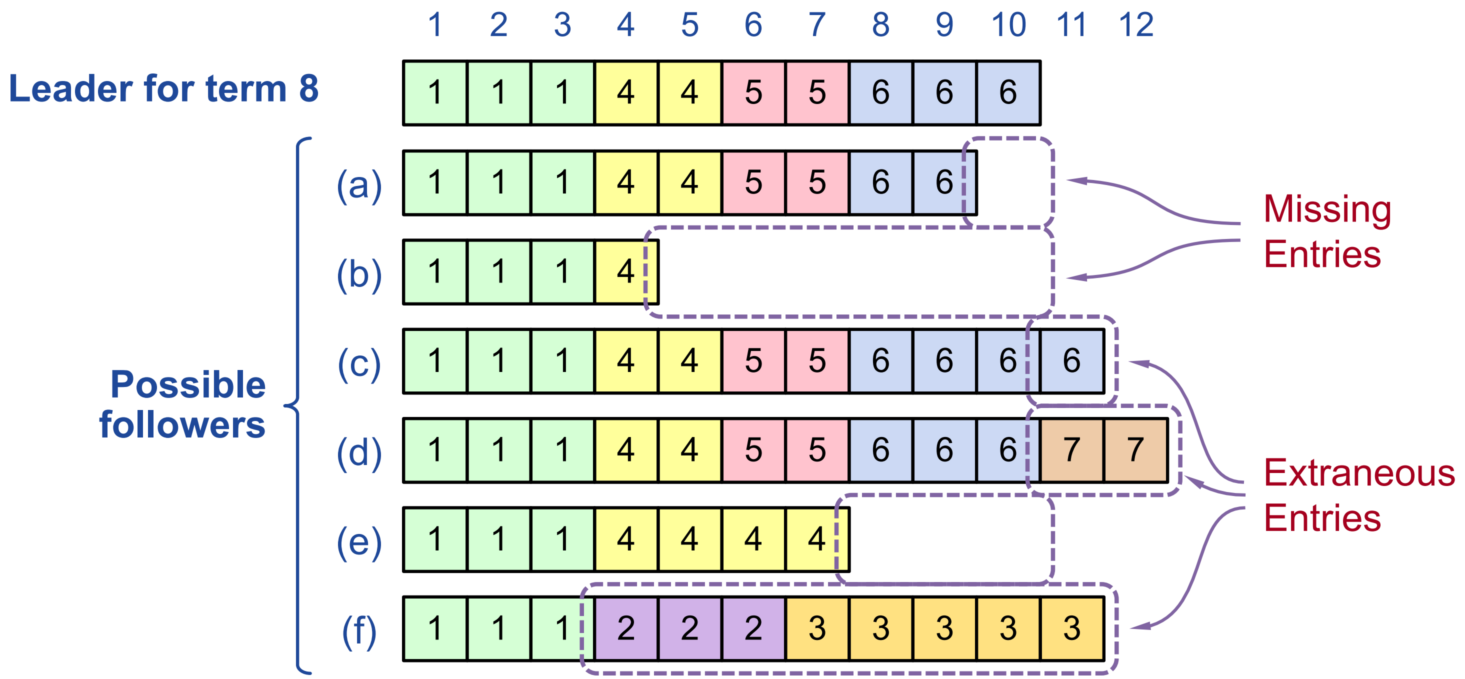


	1	2	3	4	5
s_1	1	1	2	4	
s_2	1	1	2	4	
s_3	1	1	2	4	
s_4	1	1			
s_5	1	1	3	3	3

Leader for term 4

- For leader to decide entry is committed:
  1. Entry stored on a majority
  2. ≥ 1 new entry from leader's term also on majority
- Example; Once e4 committed, s_5 cannot be elected leader for term 5, and e3 and e4 both safe

Challenge: Log Inconsistencies



	1	2	3	4	5	6	7	8	9	10	11	12
Leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

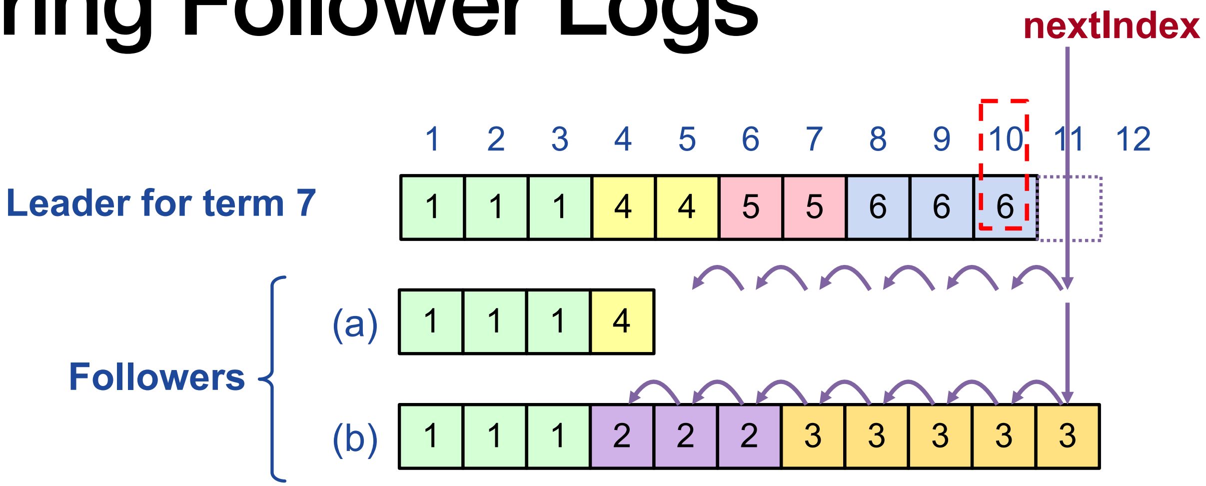
Possible followers

Missing Entries

Extraneous Entries

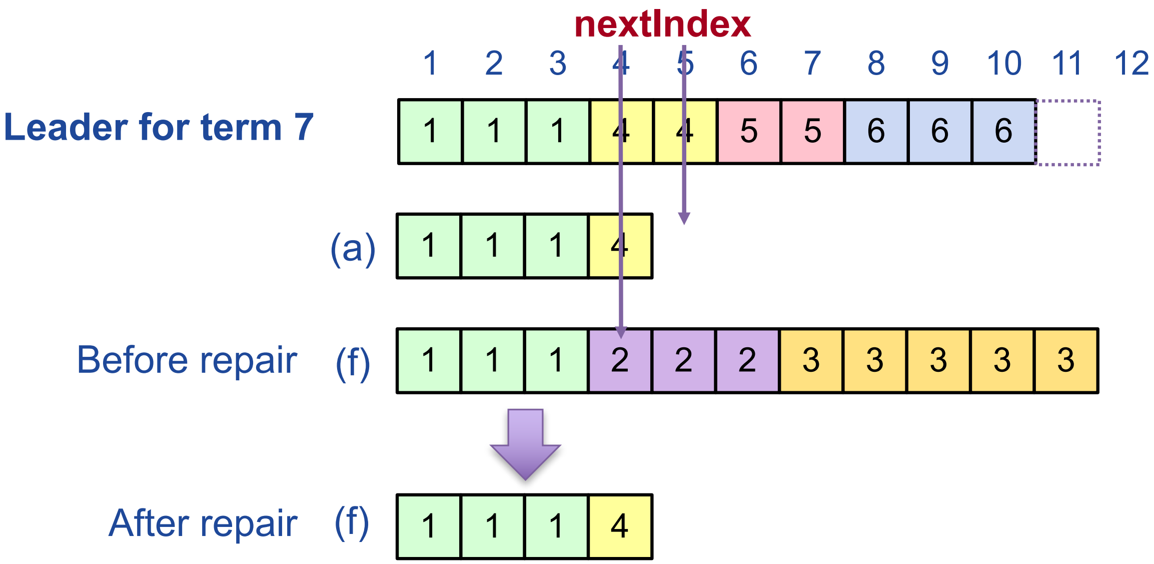
Leader changes can result in log inconsistencies

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)
- If AppendEntries consistency check fails, decrement nextIndex, try again

Repairing Follower Logs



Neutralizing Old Leaders

Leader temporarily disconnected

- → other servers elect new leader
  - → old leader reconnected
    - → old leader attempts to commit log entries

- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - Sender's term < receiver:
    - Receiver: Rejects RPC (via ACK which sender processes…)
  - Receiver's term < sender:
    - Receiver reverts to follower, updates term, processes RPC
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

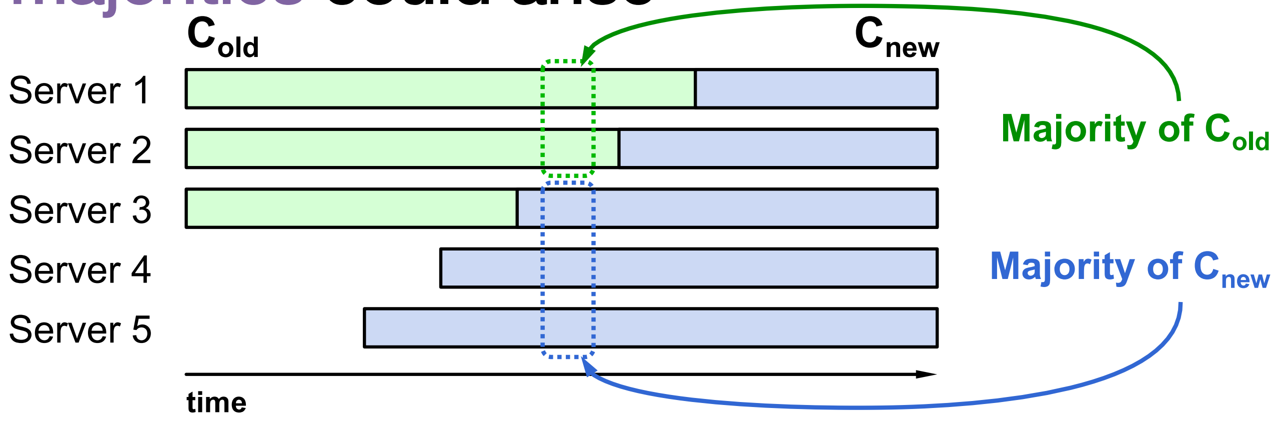
Client Protocol

- Send commands to leader
  - If leader unknown, contact any server, which redirects client to leader
- Leader only responds after command logged, committed, and executed by leader
- If request times out (e.g., leader crashes):
  - Client reissues command to new leader (after possible redirect)
- Ensure exactly-once semantics even with leader failures
  - E.g., Leader can execute command then crash before responding
  - Client should embed unique request ID in each command
  - This unique request ID included in log entry
  - Before accepting request, leader checks log for entry with same id

RECONFIGURATION

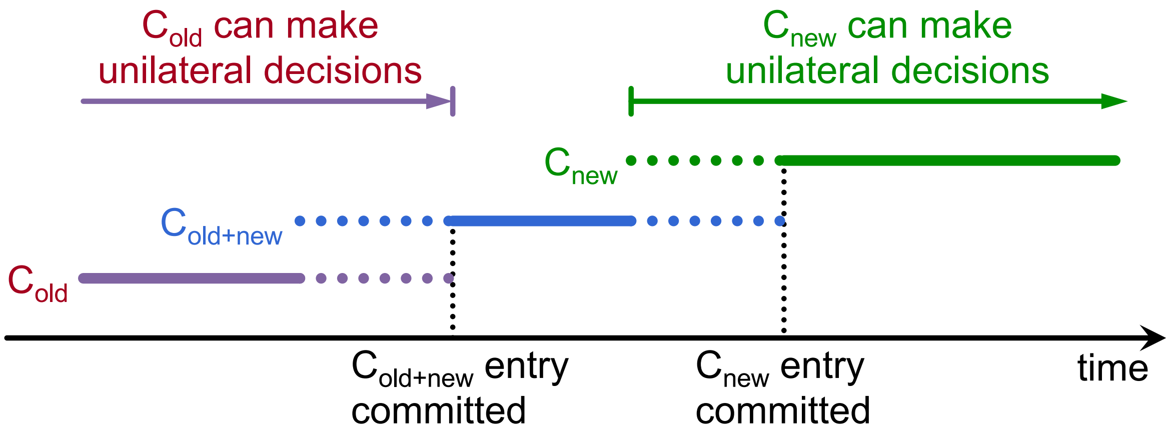
Configuration Changes

- View configuration: { leader, { members }, settings }
- Consensus must support changes to configuration
  - – Replace failed machine
  - – Change degree of replication
- Cannot switch directly from one config to another: conflicting majorities could arise



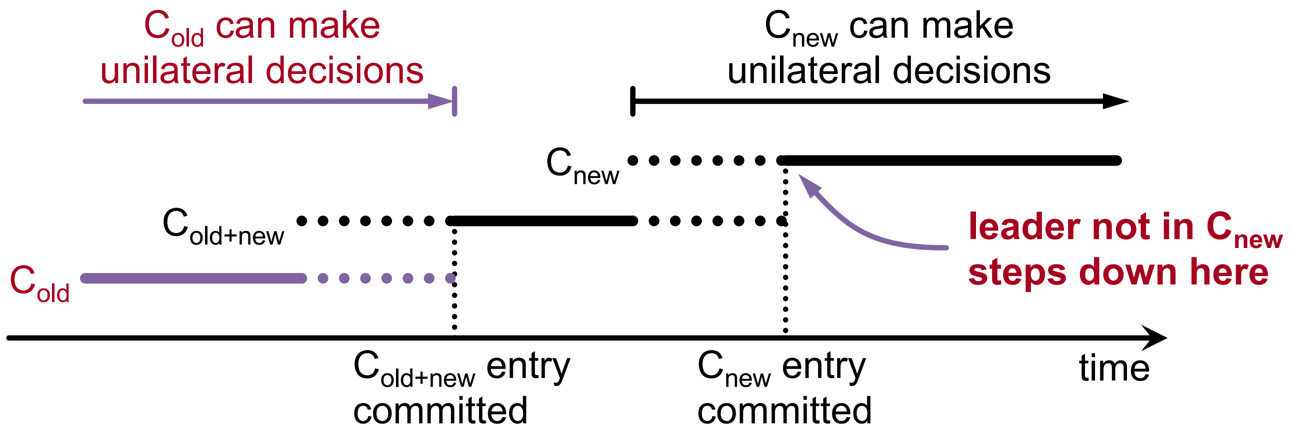
2-Phase Approach via Joint Consensus

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



2-Phase Approach via Joint Consensus

- Any server from either configuration can serve as leader
- If leader not in C_{new}, must step down once C_{new} committed



Viewstamped Replication:

A new primary copy method to support highly-available distributed systems

Oki and Liskov, PODC 1988

Raft vs. VR

- Strong leader
  - Log entries flow only from leader to other servers
  - Select leader from limited set so doesn’t need to “catch up”
- Leader election
  - Randomized timers to initiate elections
- Membership changes
  - New joint consensus approach with overlapping majorities
  - Cluster can operate normally during configuration changes