

# Putting it all together for SMR:

Two-Phase Commit, Leader Election

## RAFT



جامعة الملك عبد الله  
للعلوم والتقنية  
King Abdullah University of  
Science and Technology

---

## CS 240: Computing Systems and Concurrency Lecture 13

Marco Canini

Credits: Michael Freedman and Kyle Jamieson developed much of the original material.  
RAFT slides heavily based on those from Diego Ongaro and John Ousterhout.

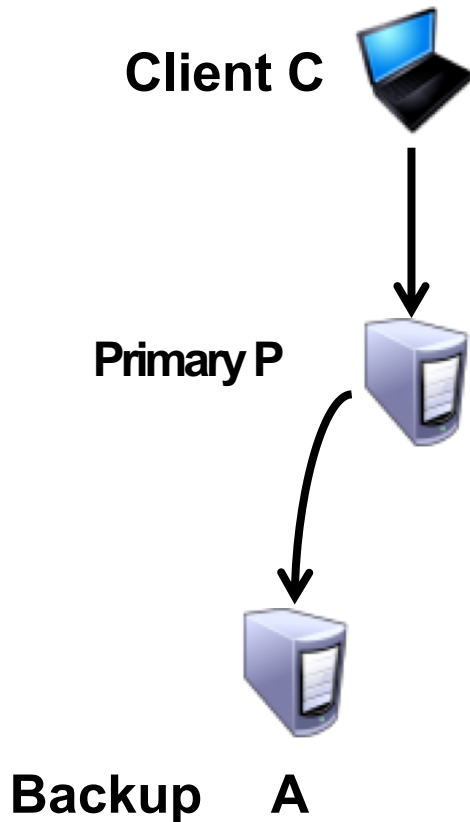
# Recall: Primary-Backup

---

- **Mechanism:** Replicate and separate servers
- **Goal #1:** Provide a highly reliable service
- **Goal #2:** Servers should behave just like a single, more reliable server

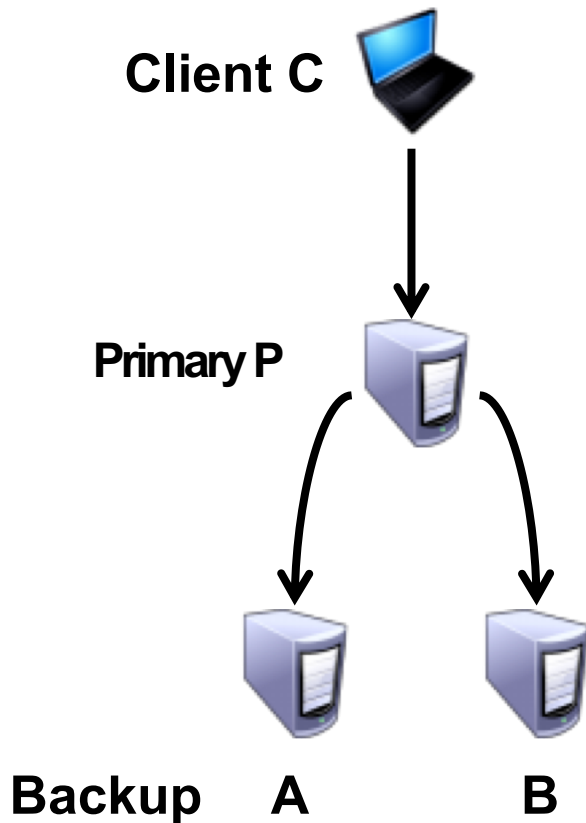
# Extend PB for high availability

---



- Primary gets ops, orders into log
  - Replicates log of ops to backup
  - Backup executes ops in same order
  - Backup takes over if primary fails
- 
- But what if network partition rather than primary failure?
    - “View” server to determine primary
    - But what if view server fails?
      - “View” determined via consensus!

# Extend PB for high availability



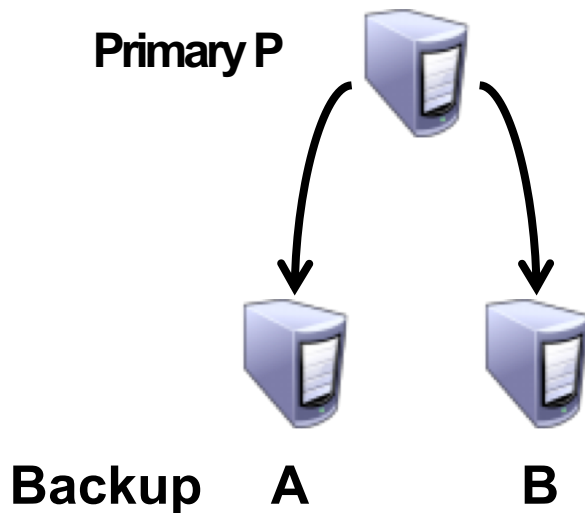
1.  $C \rightarrow P$ : “request  $\langle op \rangle$ ”
2.  $P \rightarrow A, B$ : “prepare  $\langle op \rangle$ ”
3.  $A, B \rightarrow P$ : “prepared” or “error”
4.  $P \rightarrow C$ : “result exec $\langle op \rangle$ ” or “failed”
5.  $P \rightarrow A, B$ : “commit  $\langle op \rangle$ ”

“Okay” (i.e.,  $op$  is stable) if written to  $> \frac{1}{2}$  backups

# View changes on failure

---

1. Backups monitor primary
2. If a backup thinks primary failed, initiate **View Change** (leader election)



# View changes on failure

---

1. Backups monitor primary
2. If a backup thinks primary failed, initiate **View Change** (leader election)
3. Intuitive safety argument:
  - View change requires  $f+1$  agreement
  - Op committed once written to  $f+1$  nodes
  - At least one node both saw write and in new view
4. More advanced: Adding or removing nodes (“reconfiguration”)

Requires  $2f + 1$  nodes  
to handle  $f$  failures



Backup A



Primary P

# Basic fault-tolerant Replicated State Machine (RSM) approach

1. Consensus protocol to elect leader
2. 2PC to replicate operations from leader
3. All replicas execute ops once committed

# Why bother with a leader?

Not necessary, but ...

- Decomposition: normal operation vs. leader changes
- Simplifies normal operation (no conflicts)
- More efficient than leader-less approaches
- Obvious place to handle non-determinism

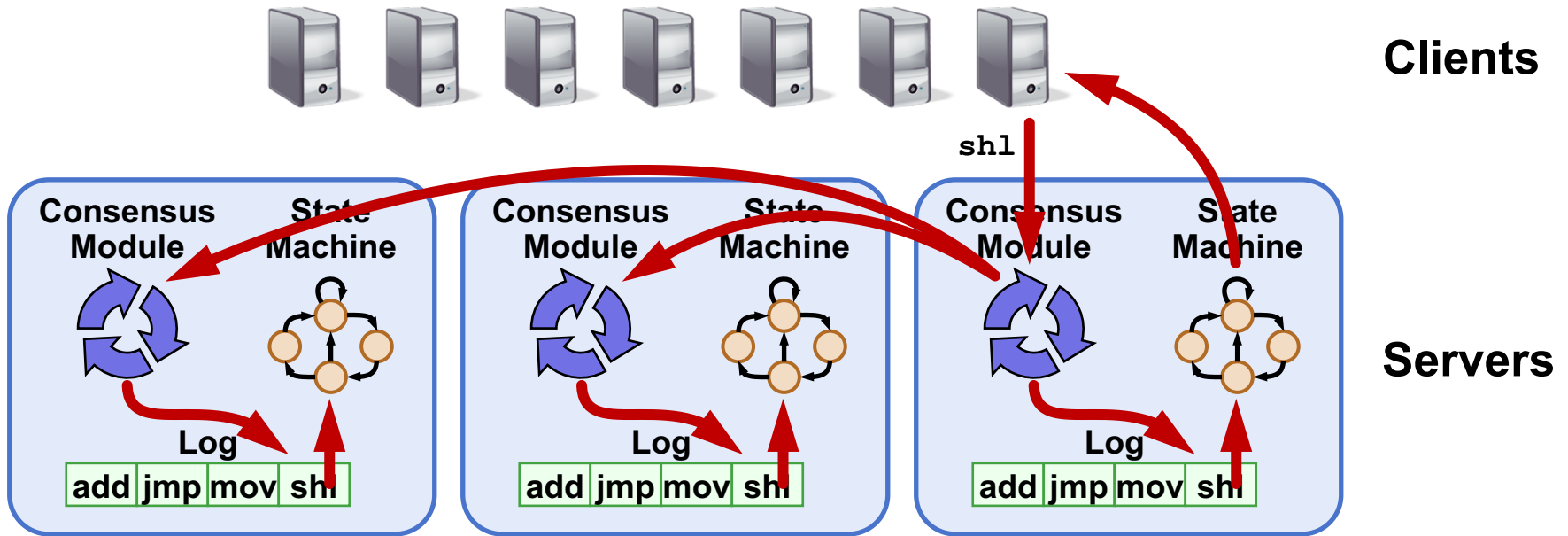


# **Raft: A Consensus Algorithm for Replicated Logs**

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

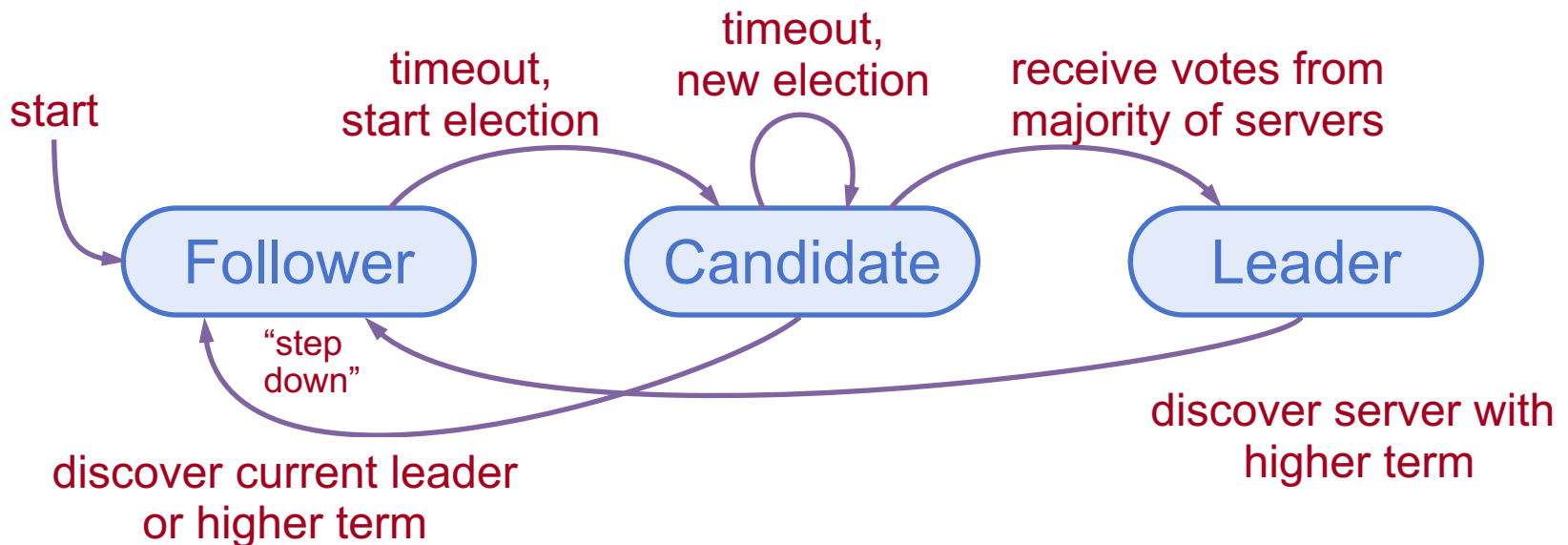
Follower

Candidate

Leader

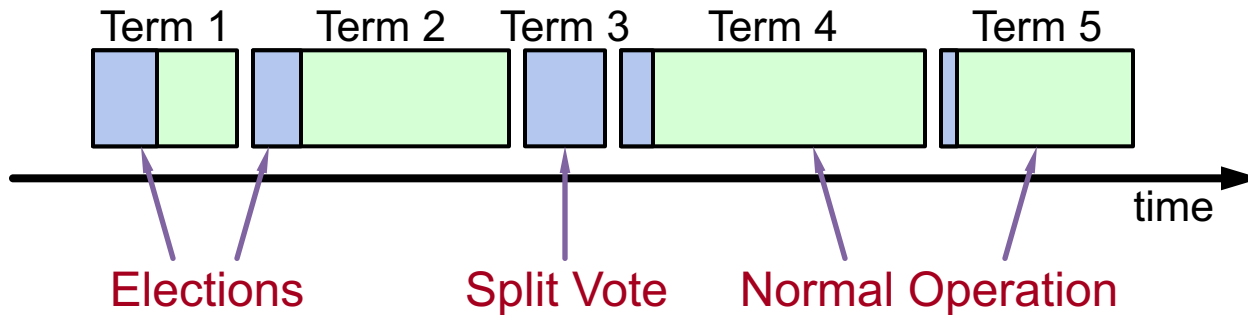
# Liveness Validation

- 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

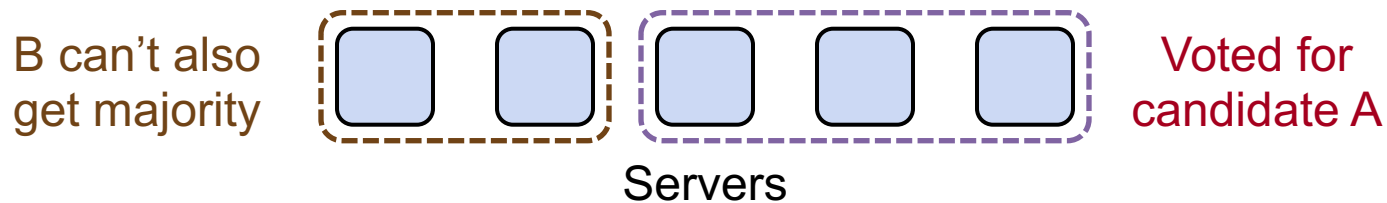
---

- **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

---

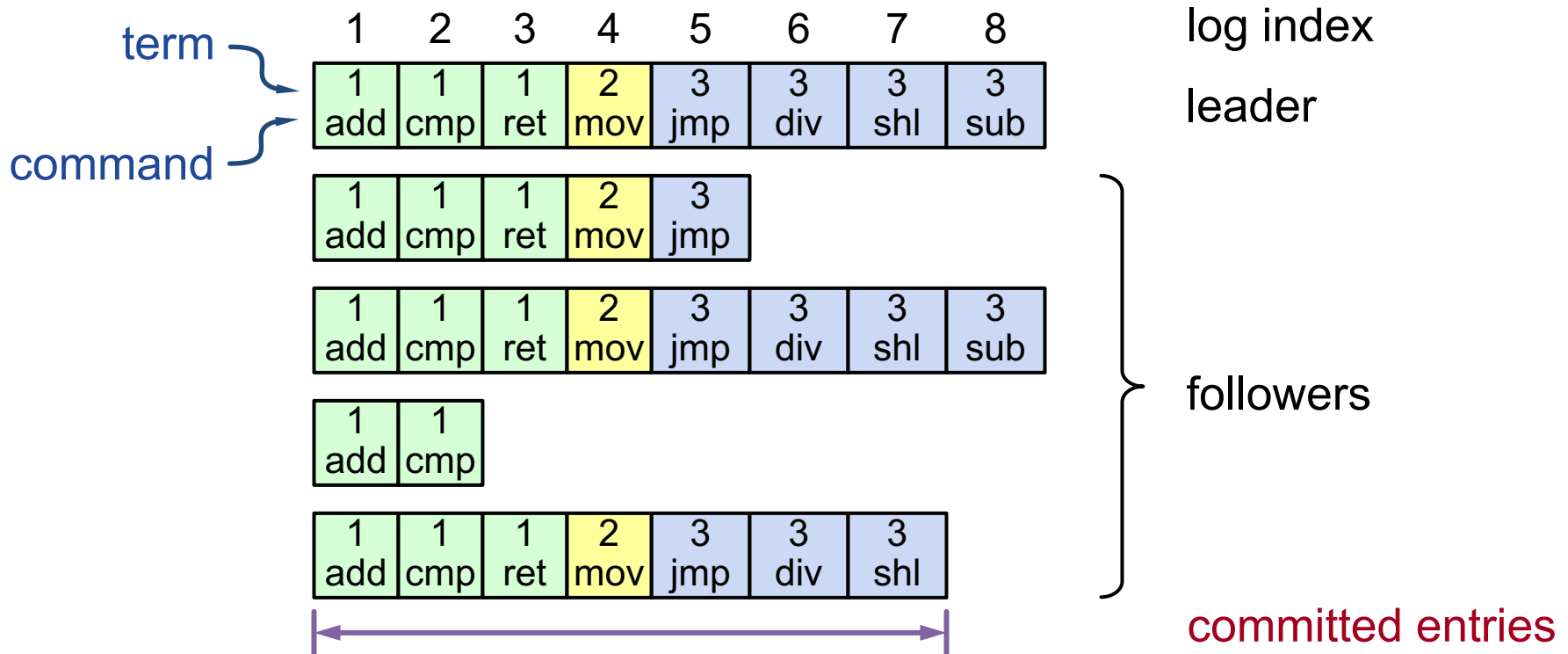
- **Safety:** allow at most one winner per term
  - Each server votes only once per term (persists on disk)
  - Two different candidates can't get majorities in same term



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

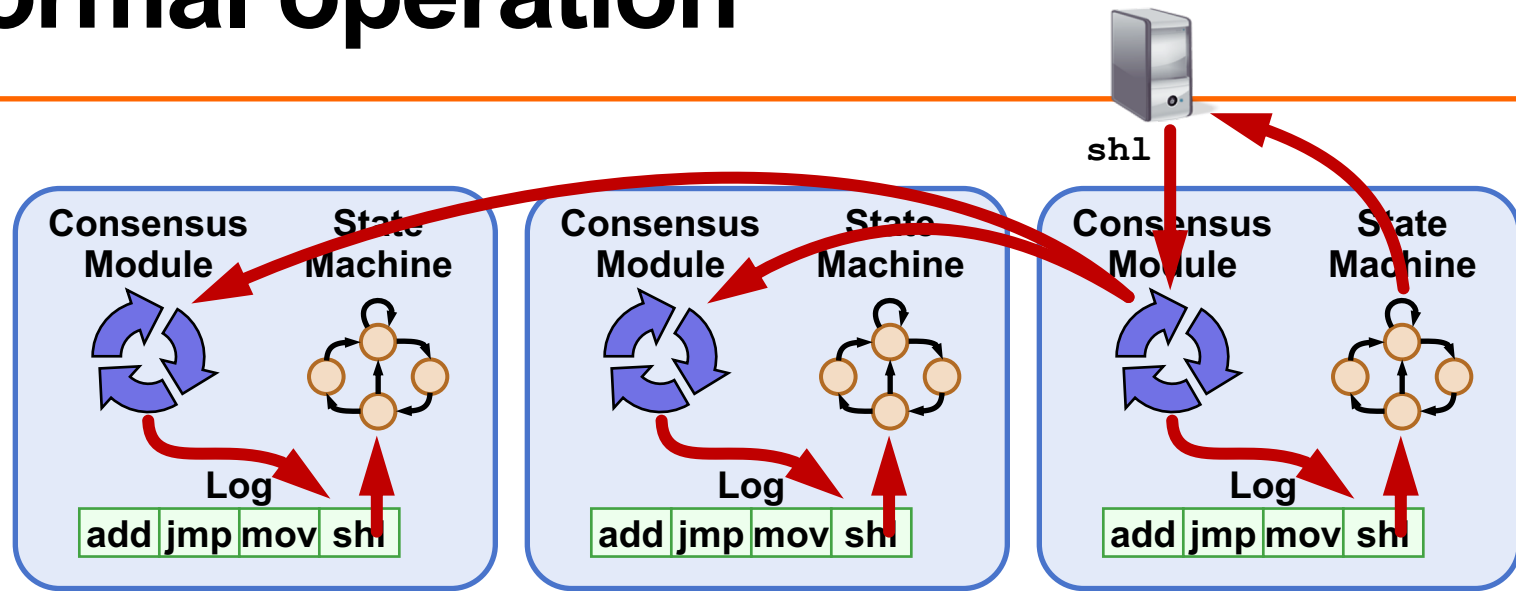


# Log Structure



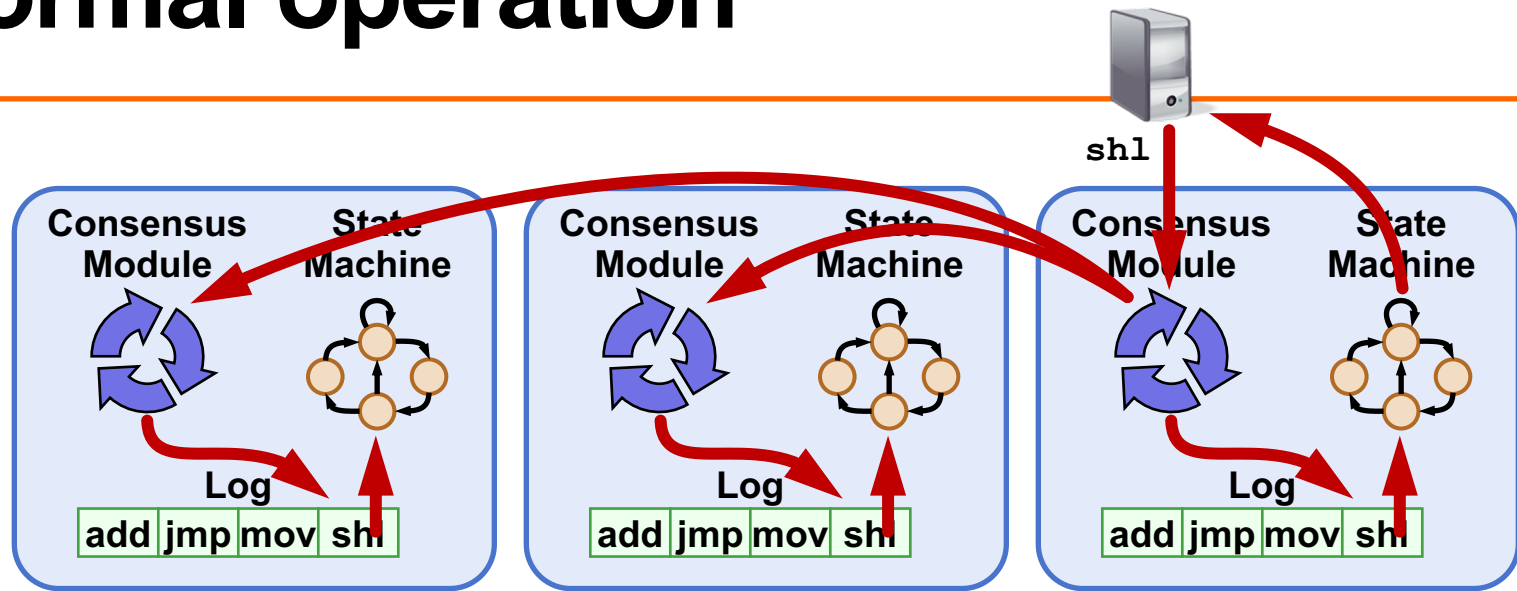
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry **committed** if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

# 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

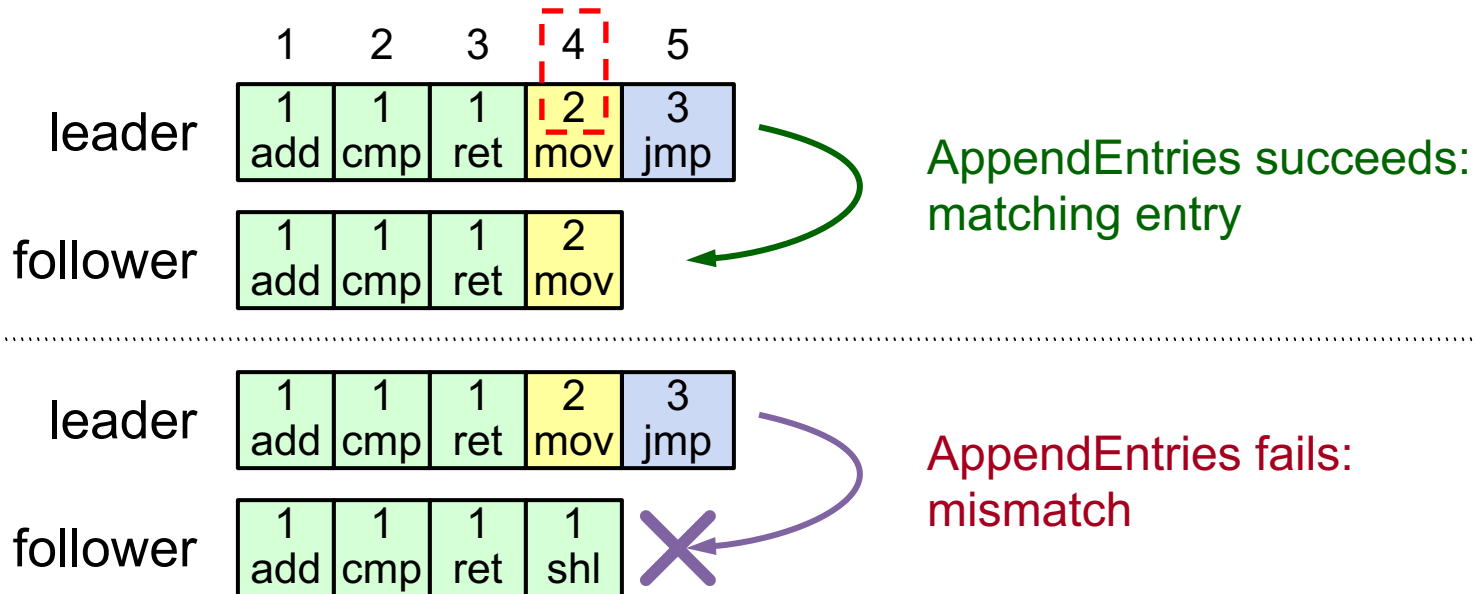
# Log Operation: Highly Coherent

---

	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

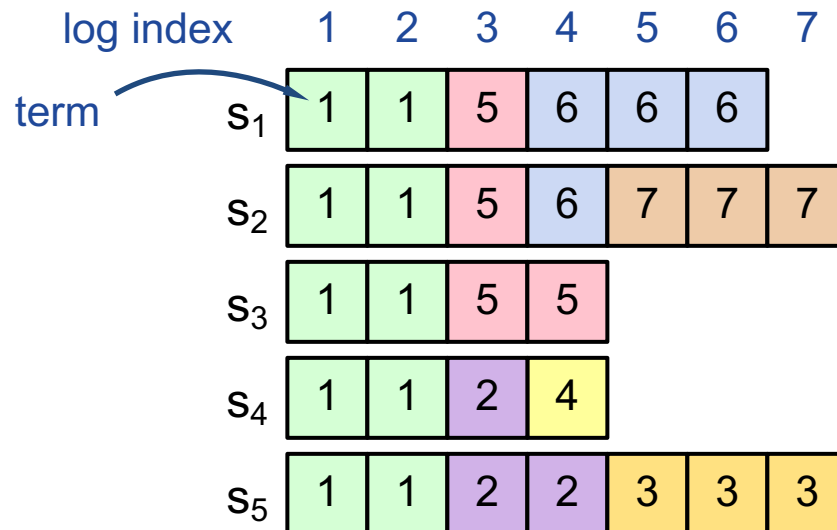
# Log Operation: Consistency Check



- AppendEntries has  $\langle \text{index}, \text{term} \rangle$  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



# Safety Requirement

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

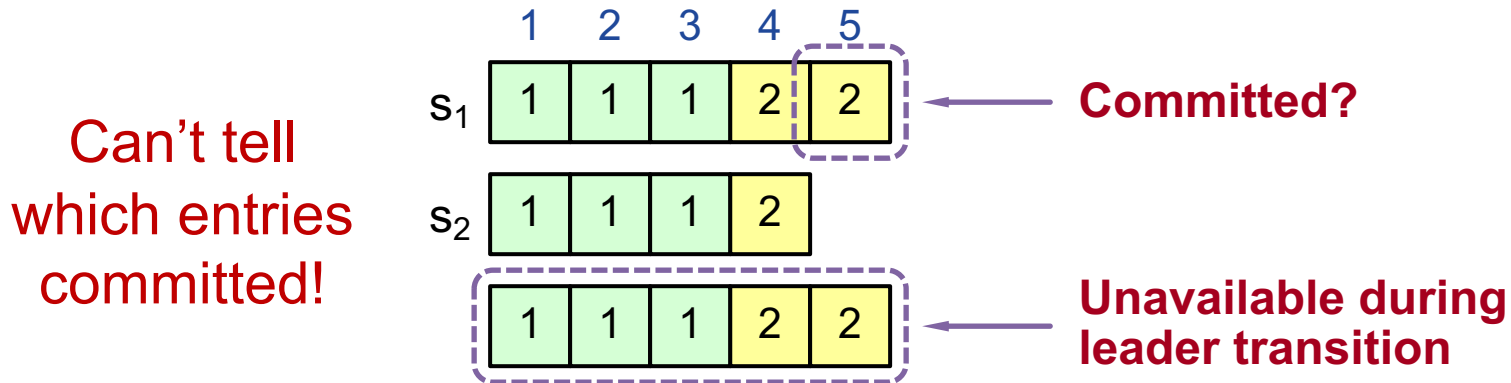
- **Raft safety property:** If leader has decided log entry is committed, entry will be present in logs of all future leaders
- **Why does this guarantee higher-level goal?**
  1. Leaders never overwrite entries in their logs
  2. Only entries in leader's log can be committed
  3. Entries must be committed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

**Committed** → **Present in future leaders' logs**

Restrictions on  
commitment

Restrictions on  
leader election

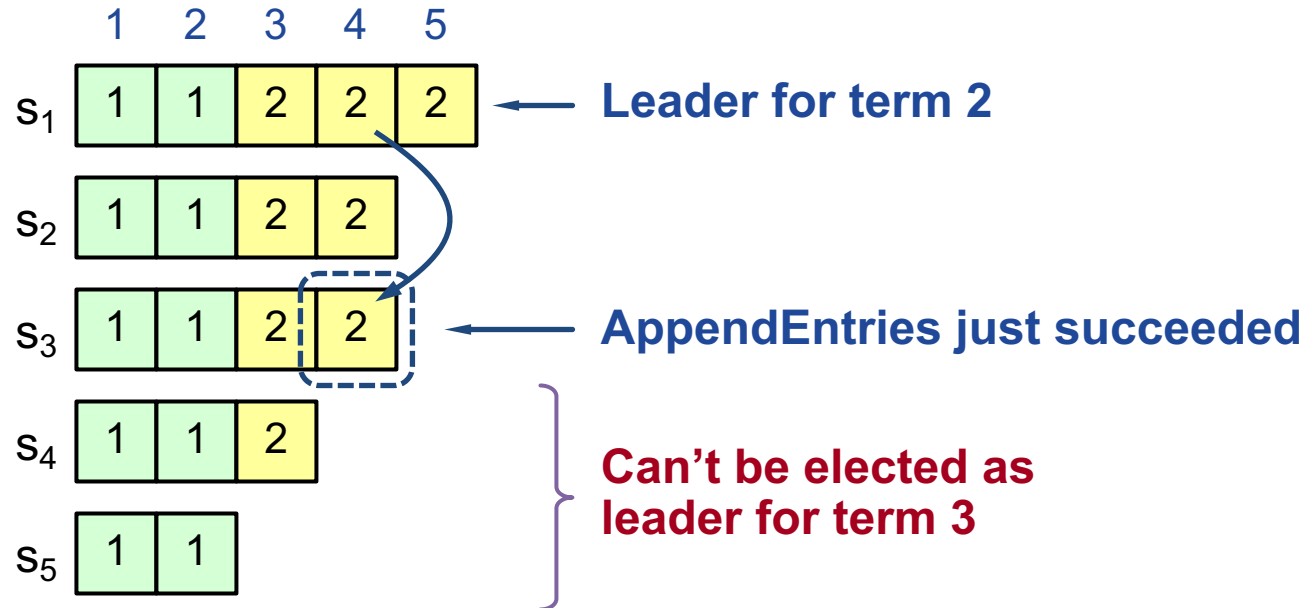
# Picking the Best Leader



- 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

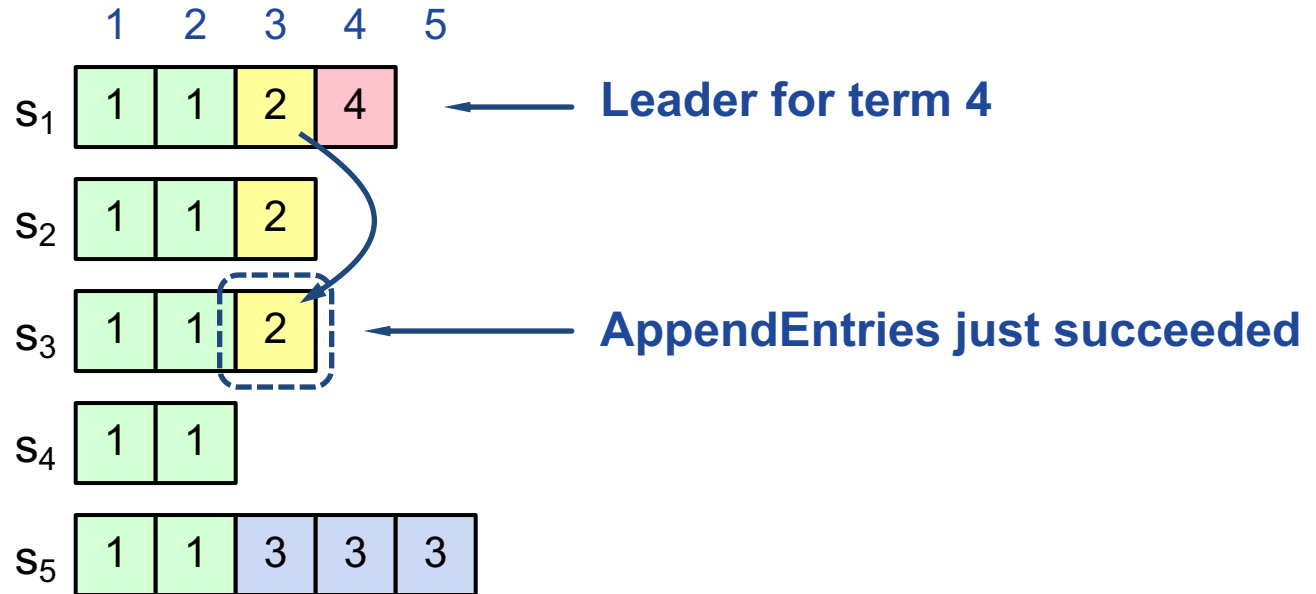


# Committing Entry from Current Term



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

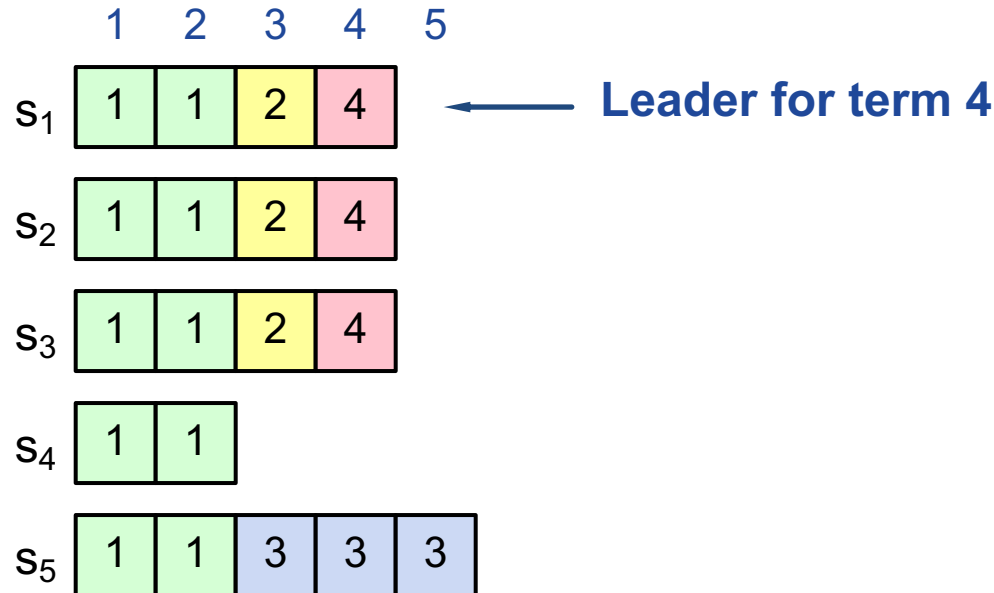
# Committing Entry from Earlier Term



- **Case #2:** Leader trying to finish committing entry from earlier
- Entry 3 **not safely committed**:
  - s<sub>5</sub> can be elected as leader for term 5 (how?)
  - If elected, it will overwrite entry 3 on s<sub>1</sub>, s<sub>2</sub>, and s<sub>3</sub>

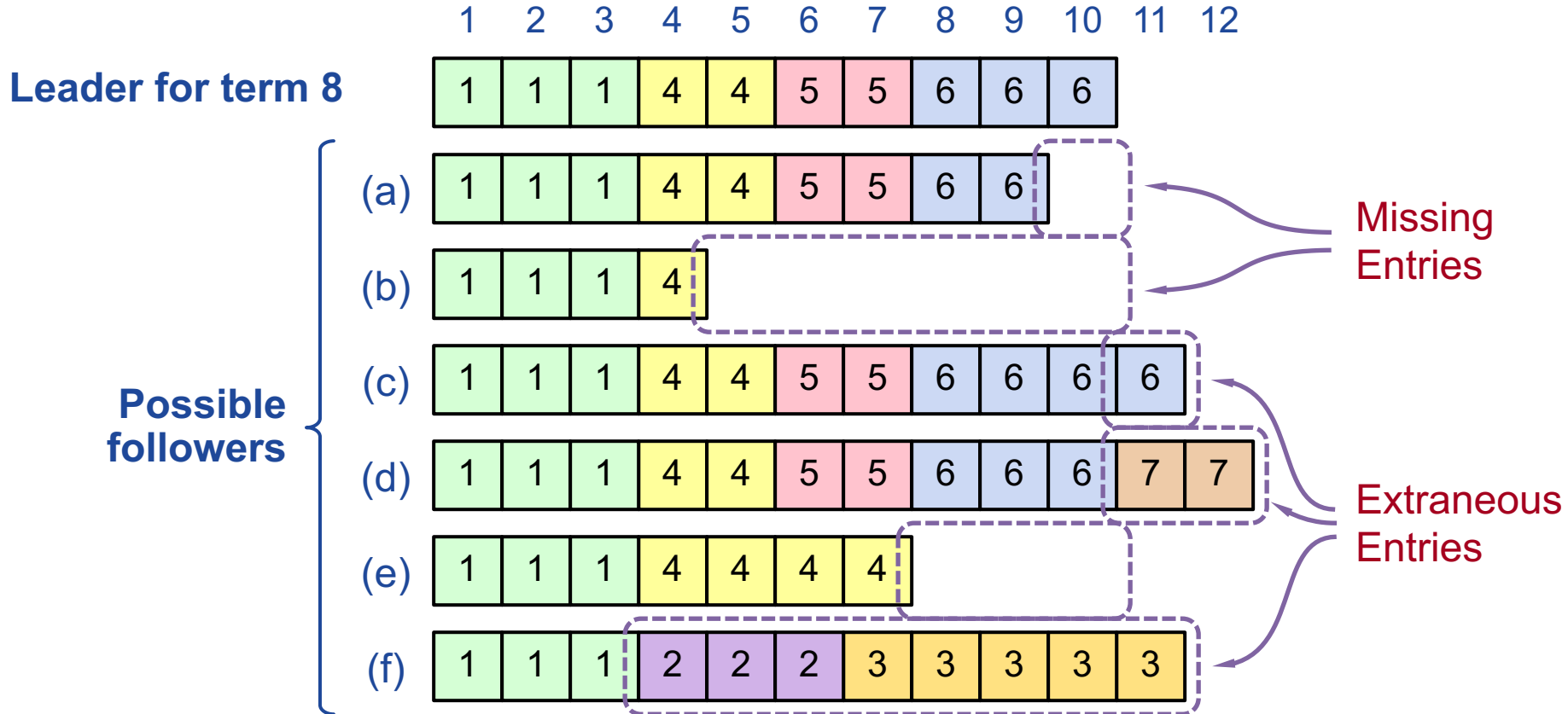
# New Commitment Rules

---



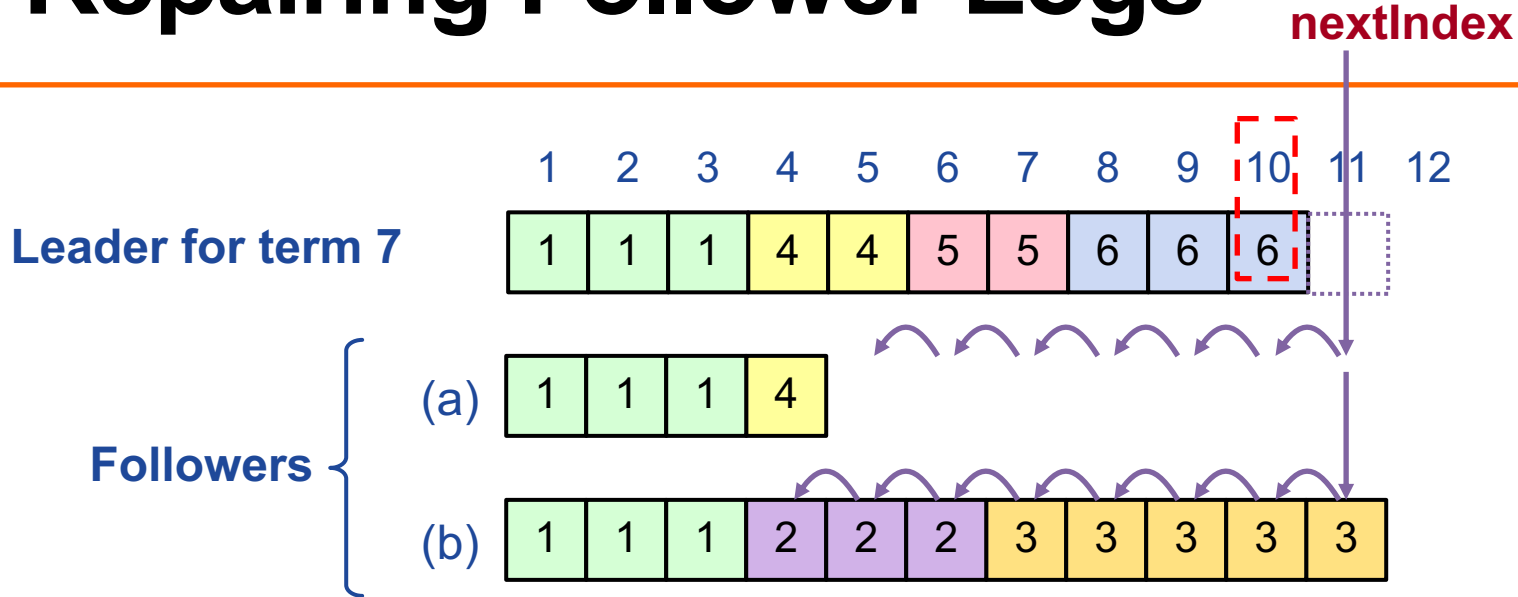
- **For leader to decide entry is committed:**
  1. Entry stored on a majority
  2.  $\geq 1$  new entry from leader's term also on majority
- Example; Once e4 committed, s<sub>5</sub> cannot be elected leader for term 5, and e3 and e4 both safe

# Challenge: Log Inconsistencies



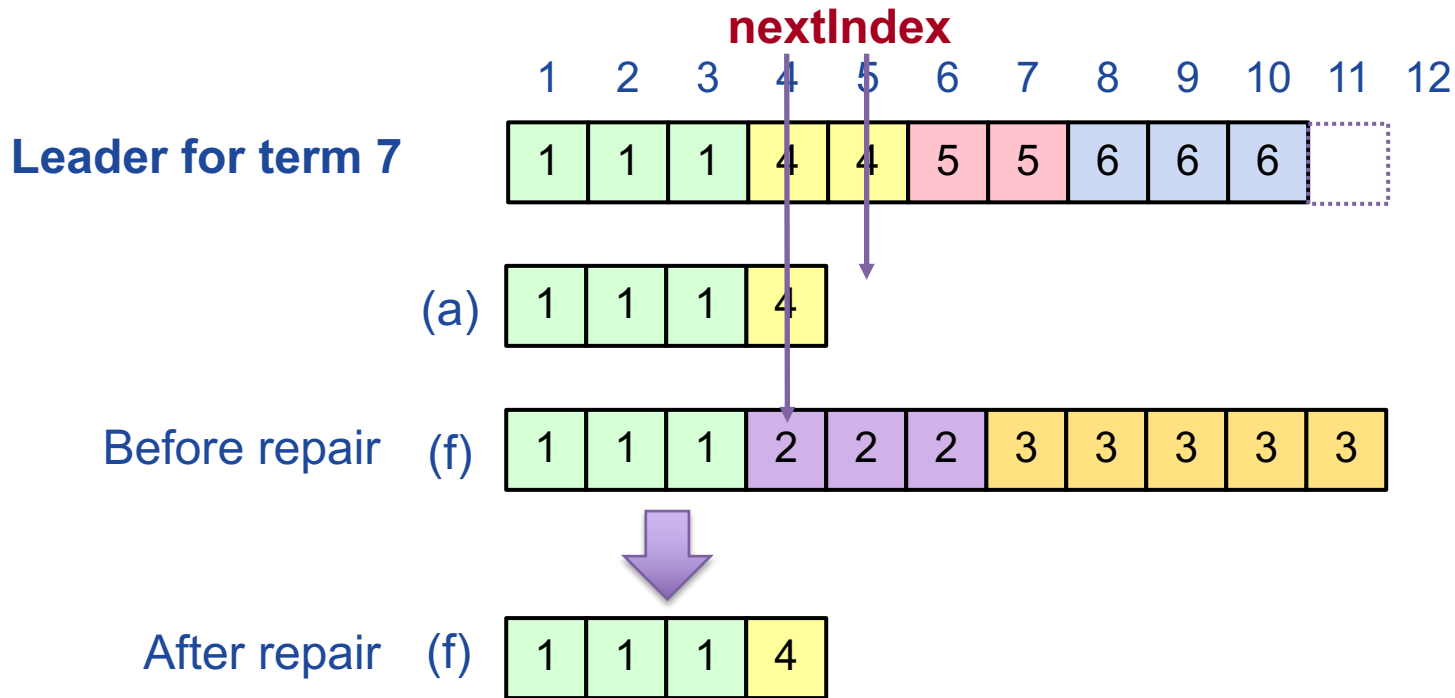
Leader changes can result in log inconsistencies

# 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  $(1 + \text{leader's last index})$
- 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

---

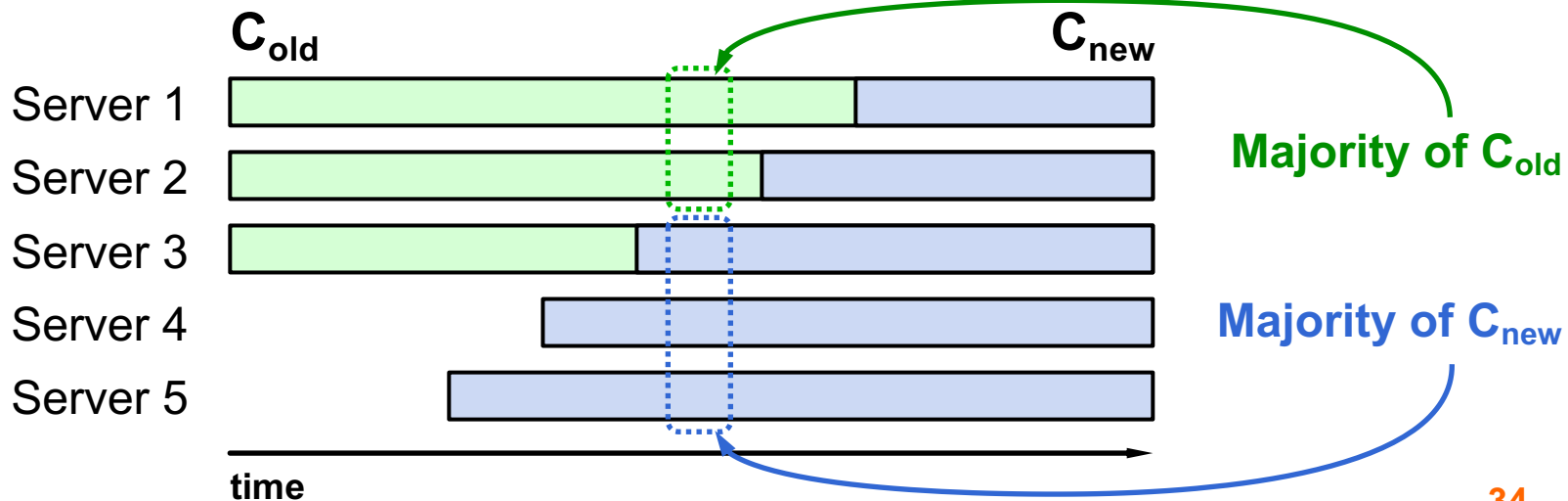
- **Send commands to leader**
  - If leader unknown, contact any server, which redirects client to leader
- **Leader only responds after command logged, committed, and executed by leader**
- **If request times out (e.g., leader crashes):**
  - Client reissues command to new leader (after possible redirect)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



# Reconfiguration

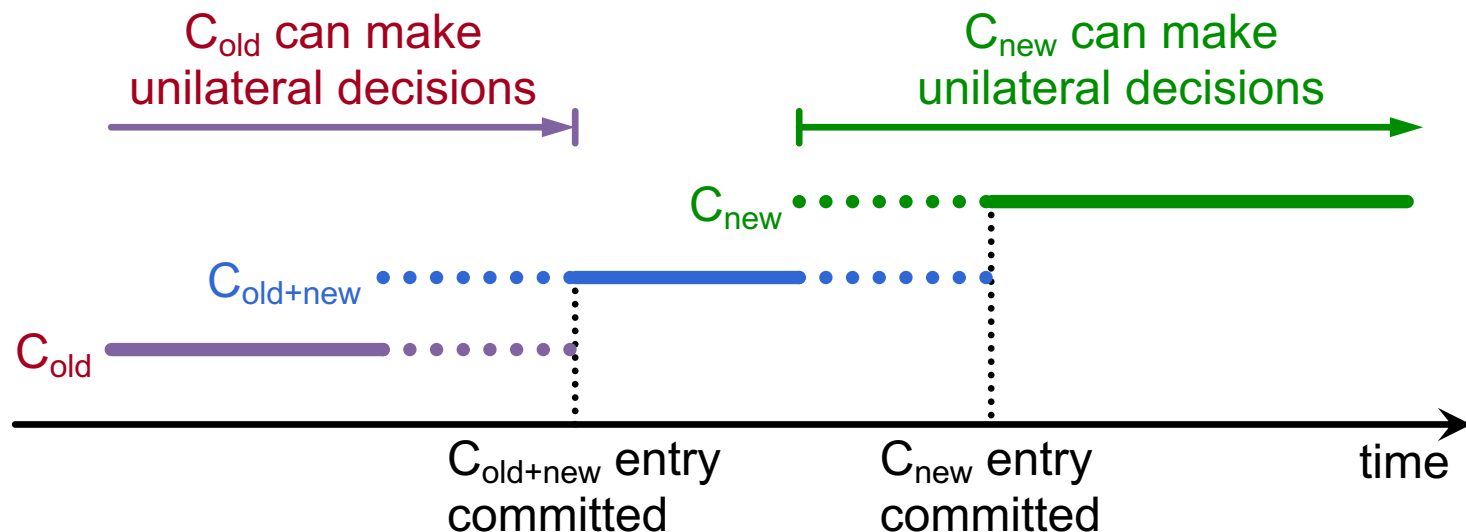
# Configuration Changes

- **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**



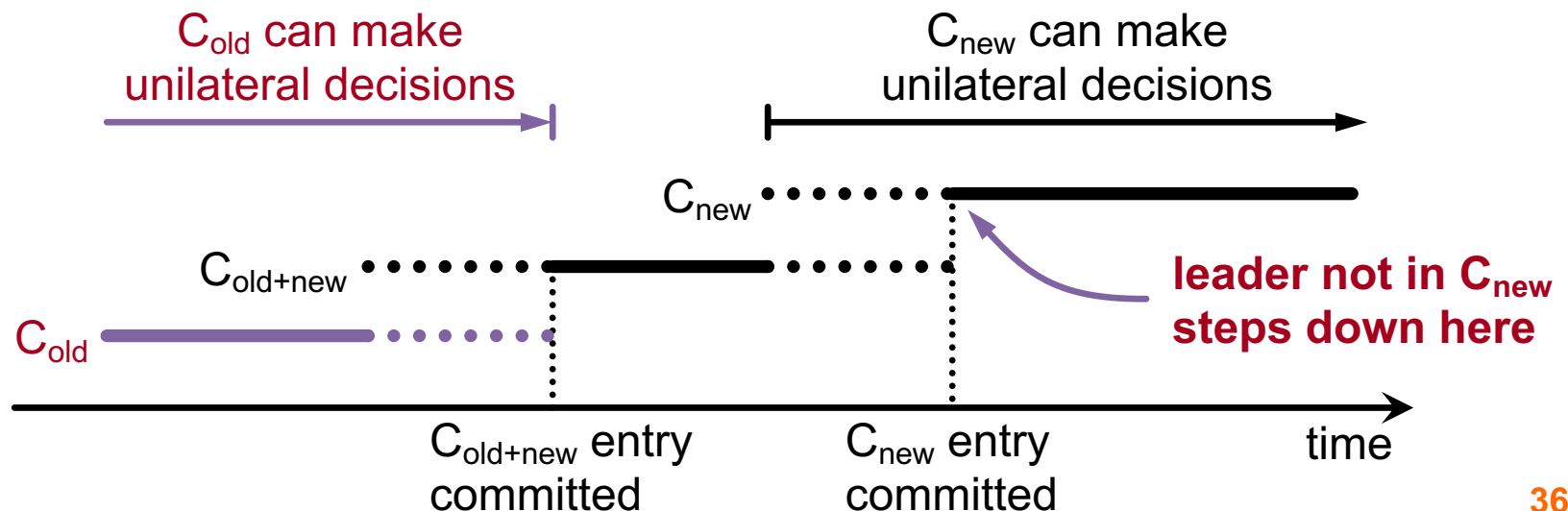
# 2-Phase Approach via Joint Consensus

- **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  $C_{\text{new}}$ , must step down once  $C_{\text{new}}$  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