Consensus II

Replicated State Machines, RAFT

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

CS 240: Computing Systems and Concurrency
Lecture 10

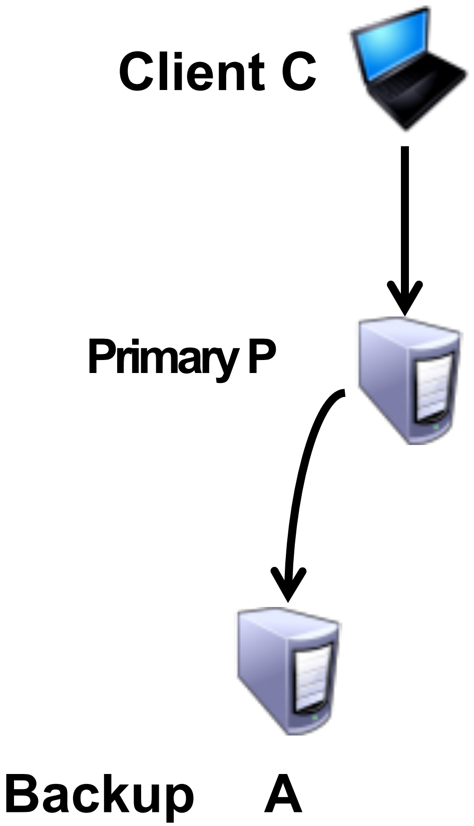
Marco Canini

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

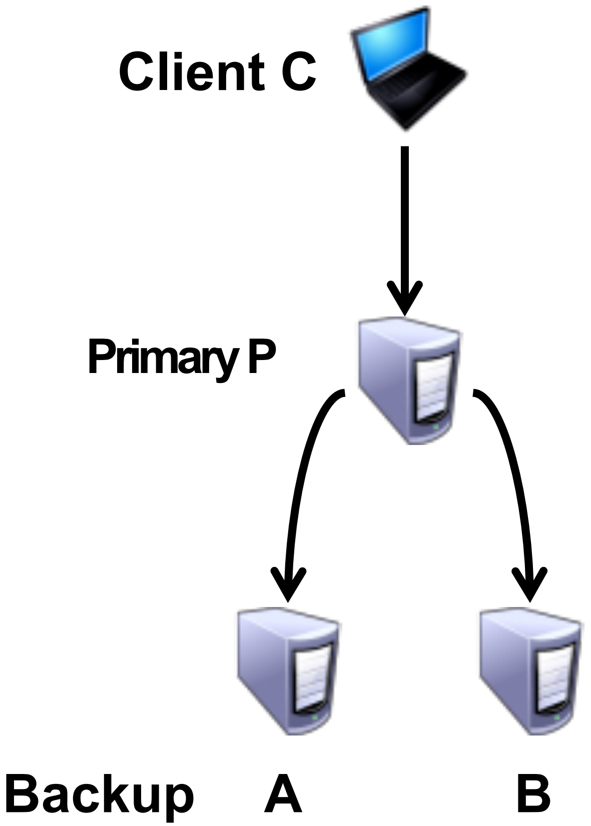
- But what if network partition rather than primary failure?
  - "View" server to determine primary
  - But what if view server fails?
    - "View" determined via consensus!

State machine replication

- **Any server** is essentially a ***state machine***
  - – Operations **transition** between states

- Need an op to be executed on all replicas, or none at all
  - – *i.e.*, we need **distributed** **all-or-nothing atomicity**
  - – If op is deterministic, replicas will end in same state

Extend PB for high availability



1. **C → P:** *"request <op>"*
2. **P → A, B:** *"prepare <op>"*
3. **A, B → P:** *"prepared"* or *"error"*
4. **P → C:** *"result exec<op>" or "failed"*
5. **P → A, B:** *"commit <op>"*

"Okay" (i.e., op is stable) if written to > ½ backups

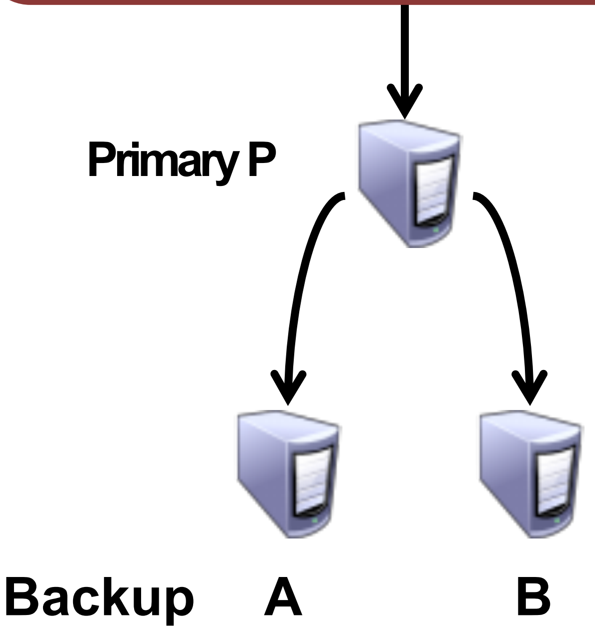
2PC from primary to backups

Expect success as replicas are all identical (unlike distributed txn)

→ P: *“request <op>”*

→ A, B: *“prepare <op>”*

3. A, B → P: *“prepared”* or *“error”*
4. P → C: *“result exec<op>”* or *“failed”*
5. P → A, B: *“commit <op>”*

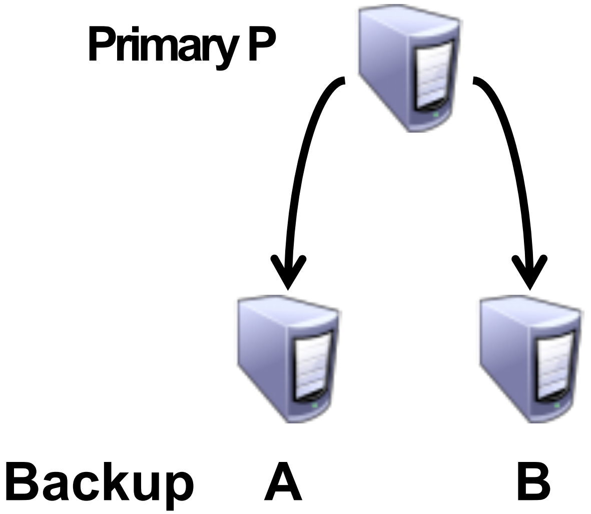


“Okay” (i.e., op is stable) if written to > ½ backups

View changes on failure

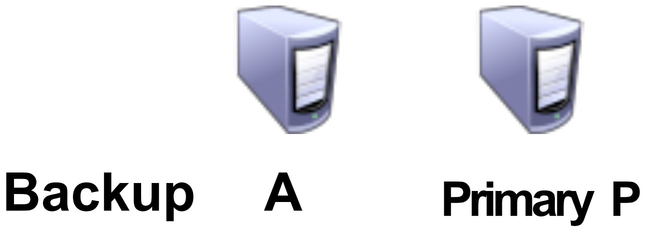
1. Backups monitor primary

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



View changes on failure

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



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”)

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 …

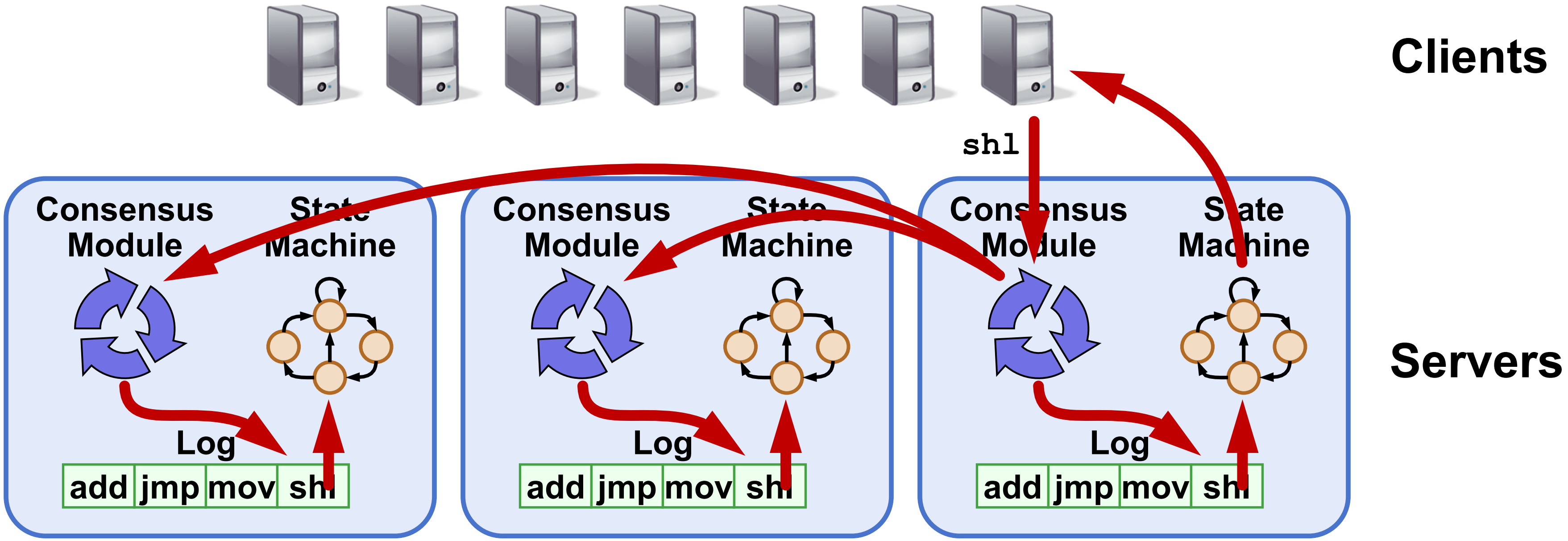
- Decomposition: normal operation vs. leader changes
- Simplifies normal operation (no conflicts)
- More efficient than leader-less approaches
- Obvious place to handle non-determinism

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

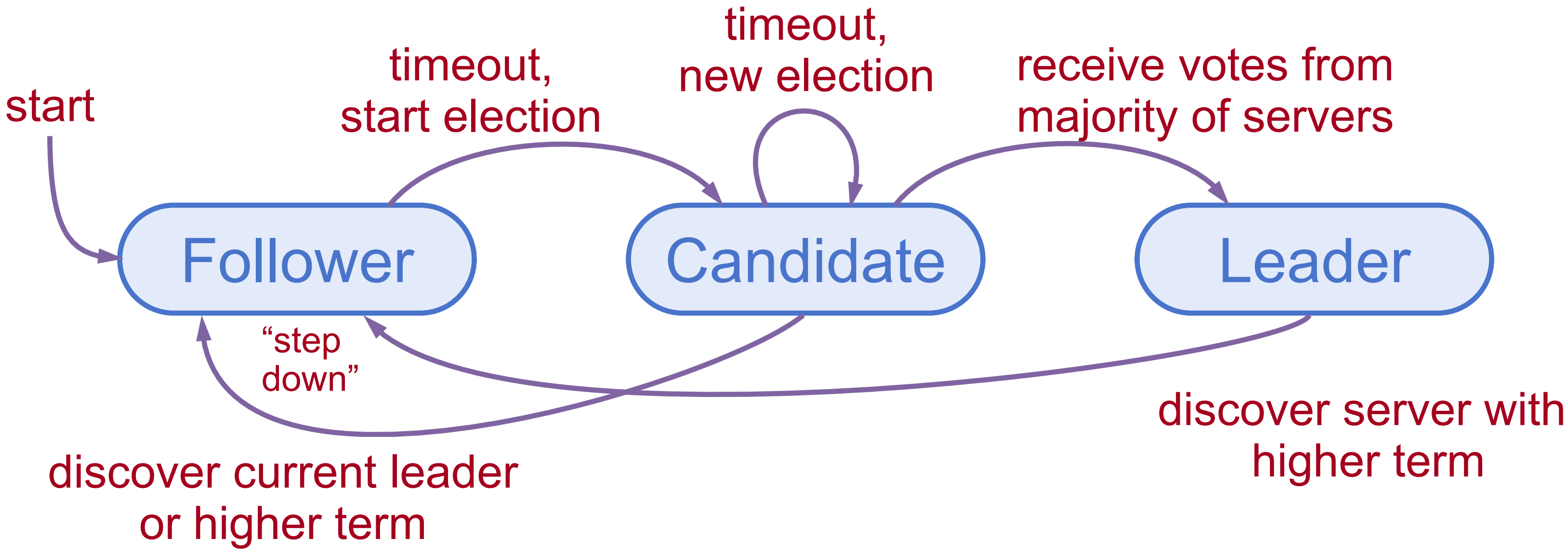
Follower

Candidate

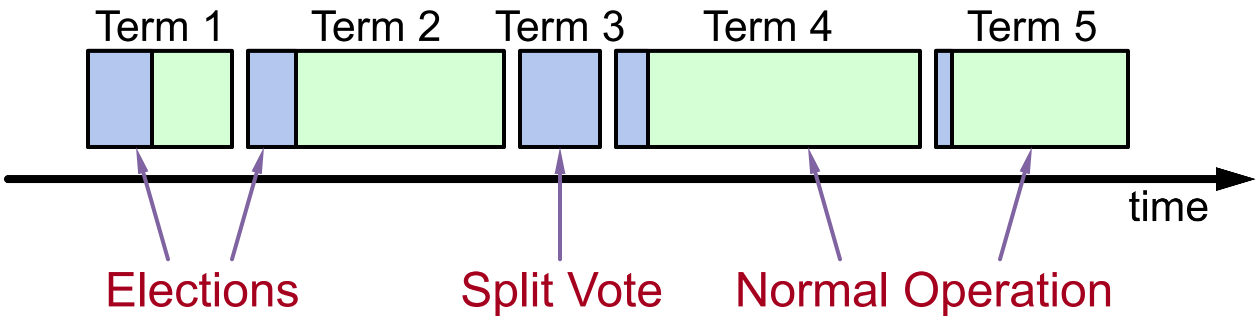
Leader

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

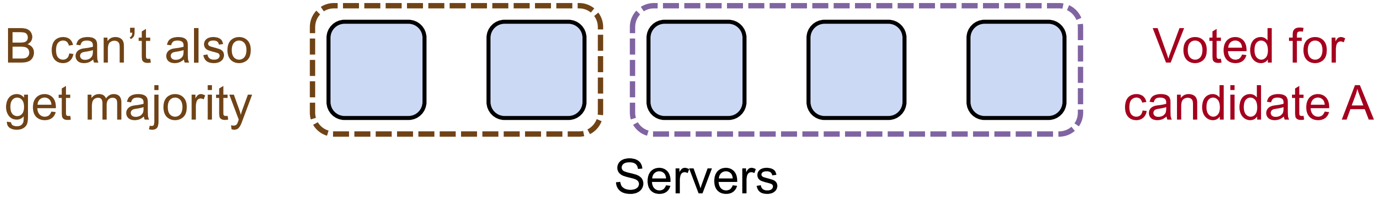
- Start election:
  - Increment current term, change to candidate state, vote for self

- 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

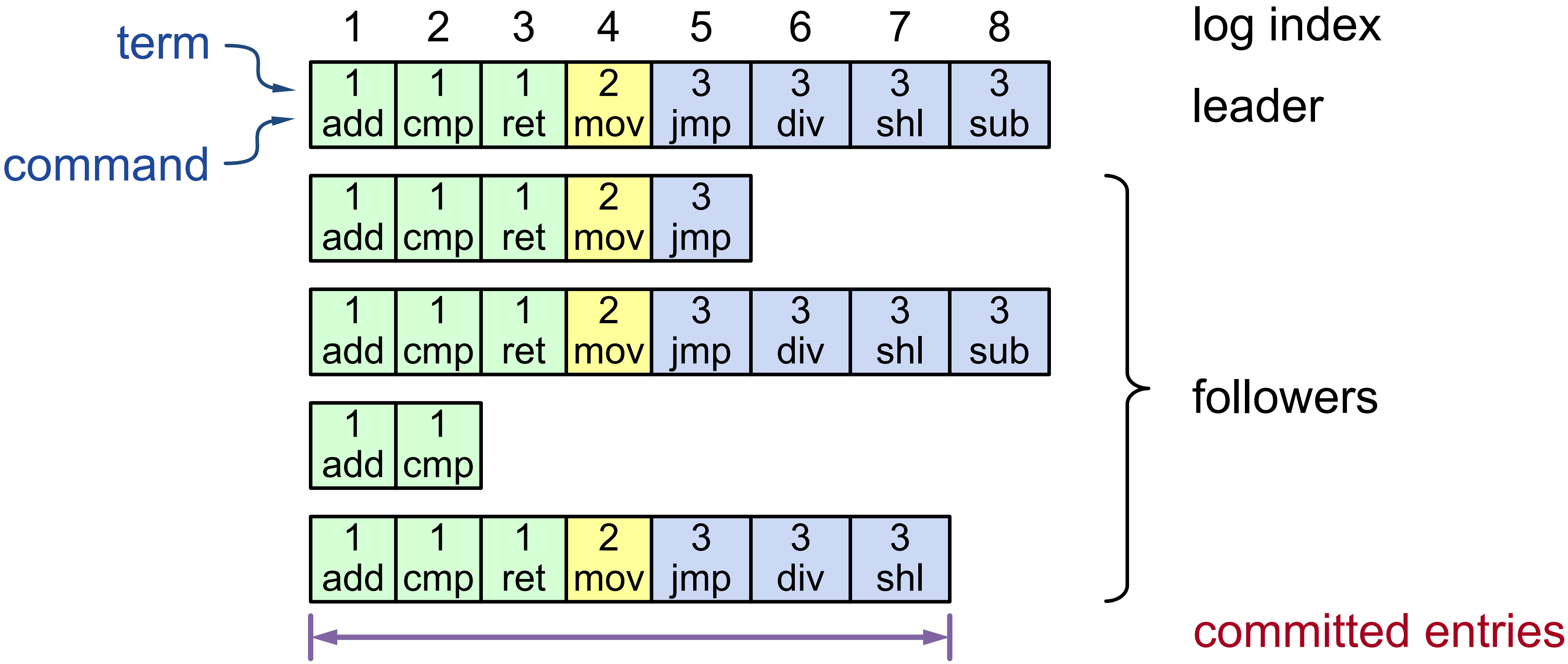
Elections

- 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 >> network RTT

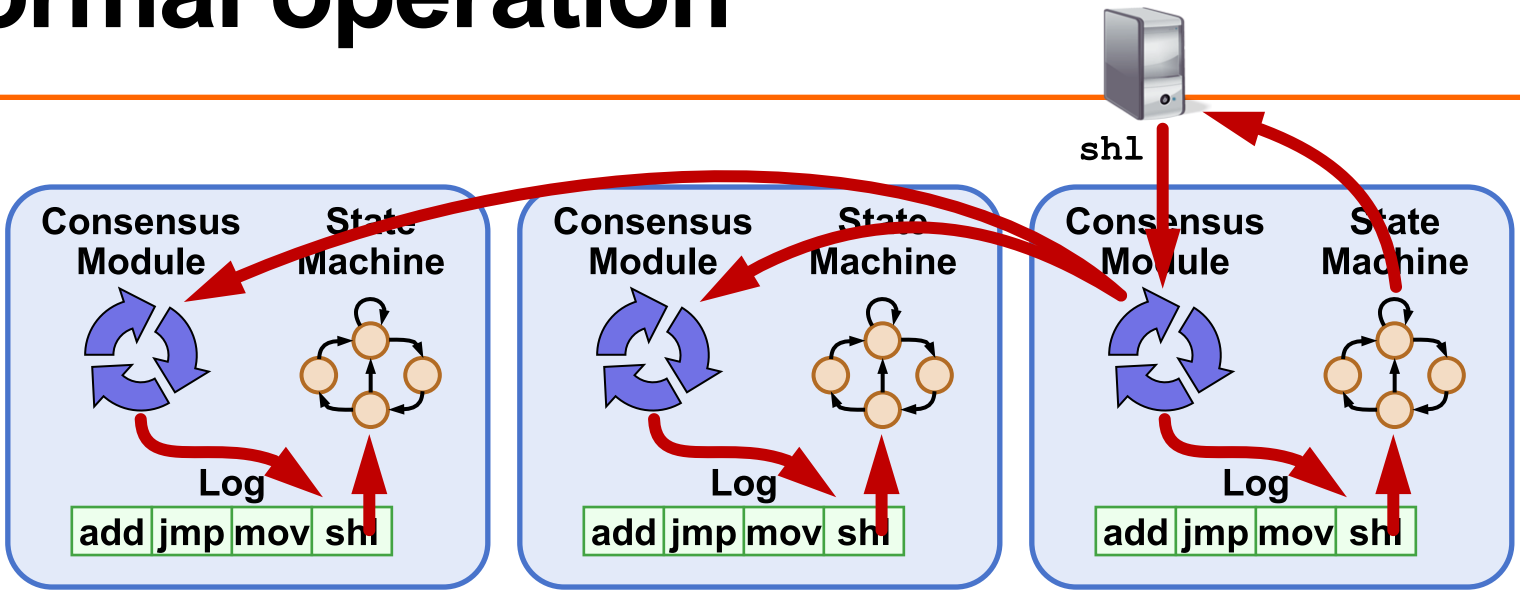
Log Structure



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	

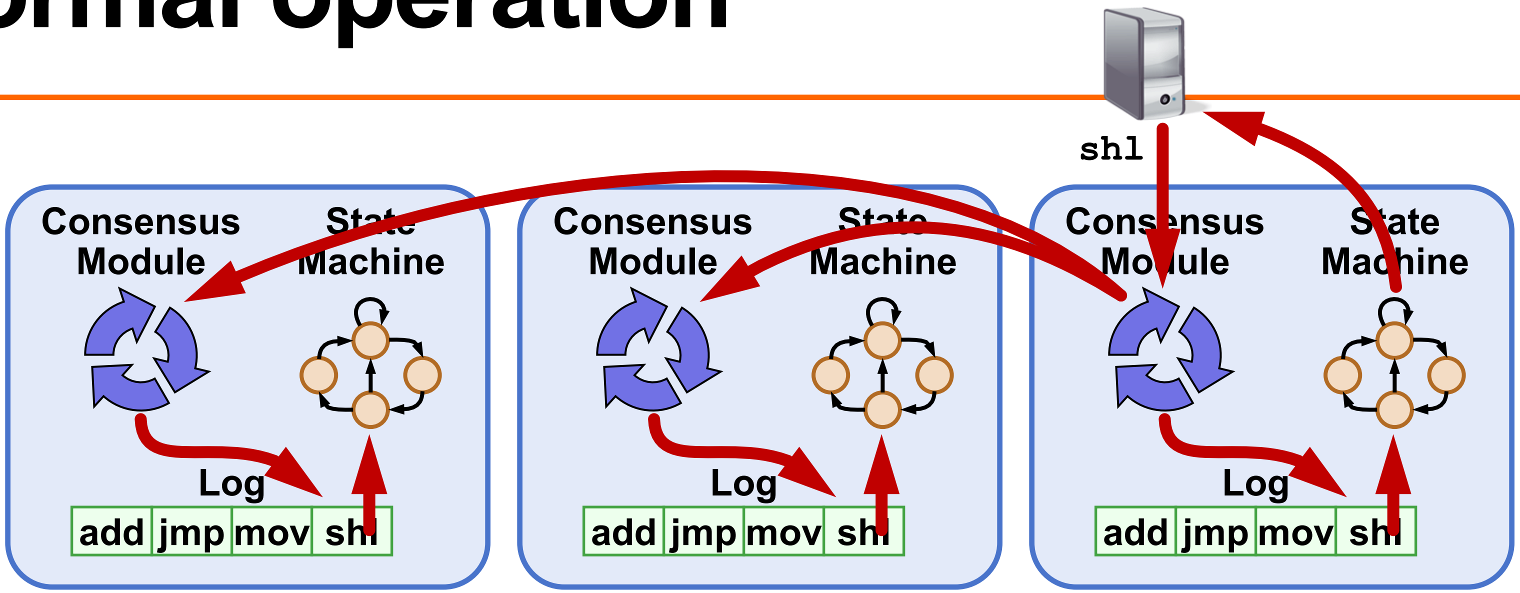
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

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

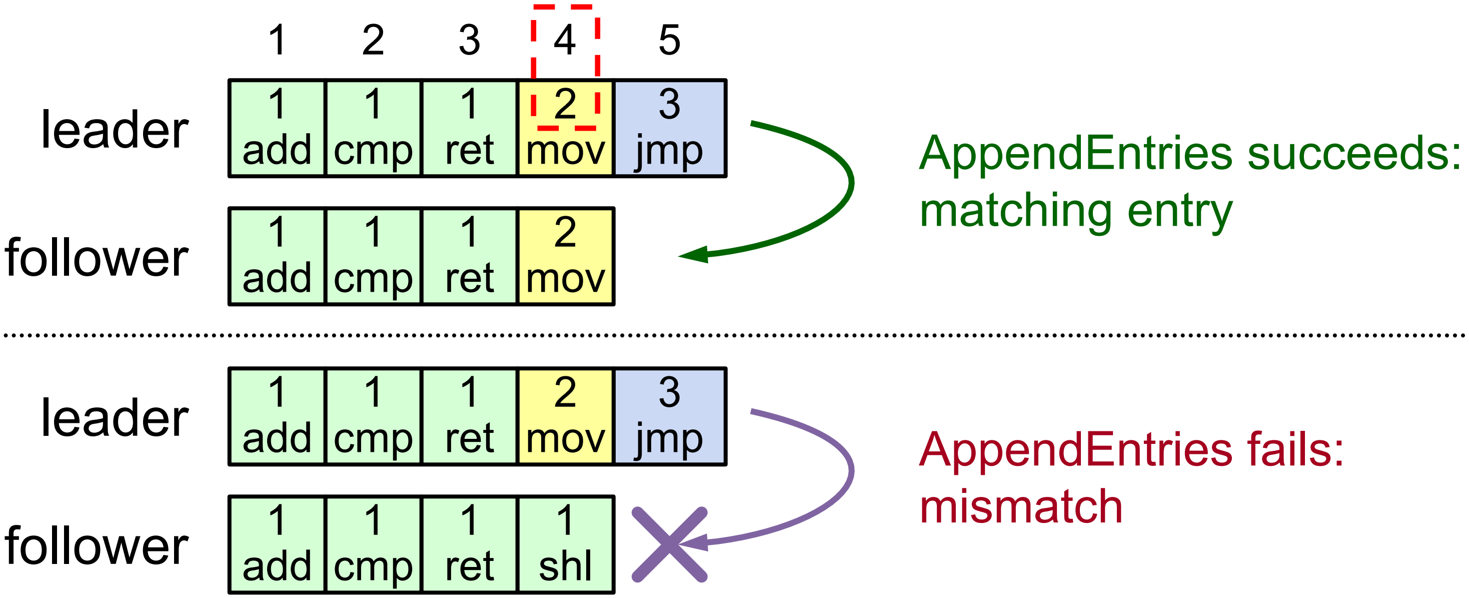
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

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



	1	2	3	4	5
leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	2 mov	

AppendEntries succeeds:
matching entry

leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	1 shl	

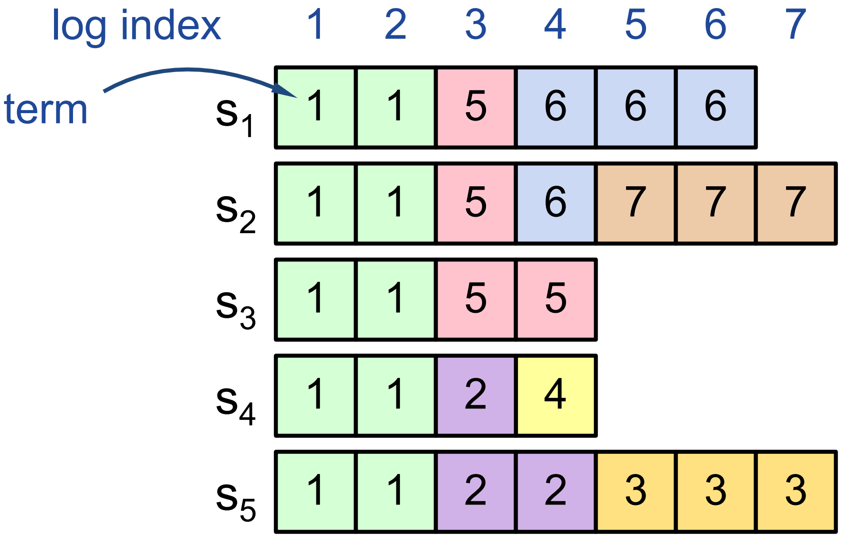
AppendEntries fails:
mismatch

- 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

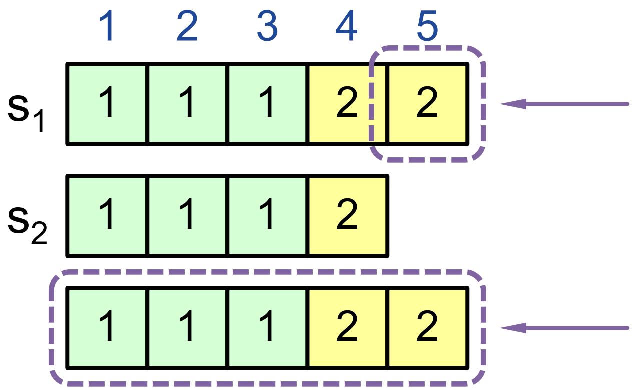
Committed → Present in future leaders' logs

Restrictions on commitment

Restrictions on leader election

Picking the Best Leader

Can’t tell which entries committed!



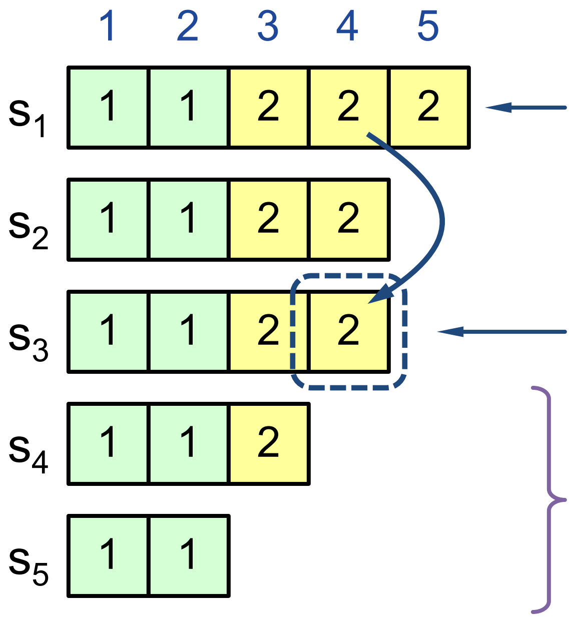
	1	2	3	4	5
s_1	1	1	1	2	2
s_2	1	1	1	2	
	1	1	1	2	2

Committed?

Unavailable during leader transition

- 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



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

Leader for term 2

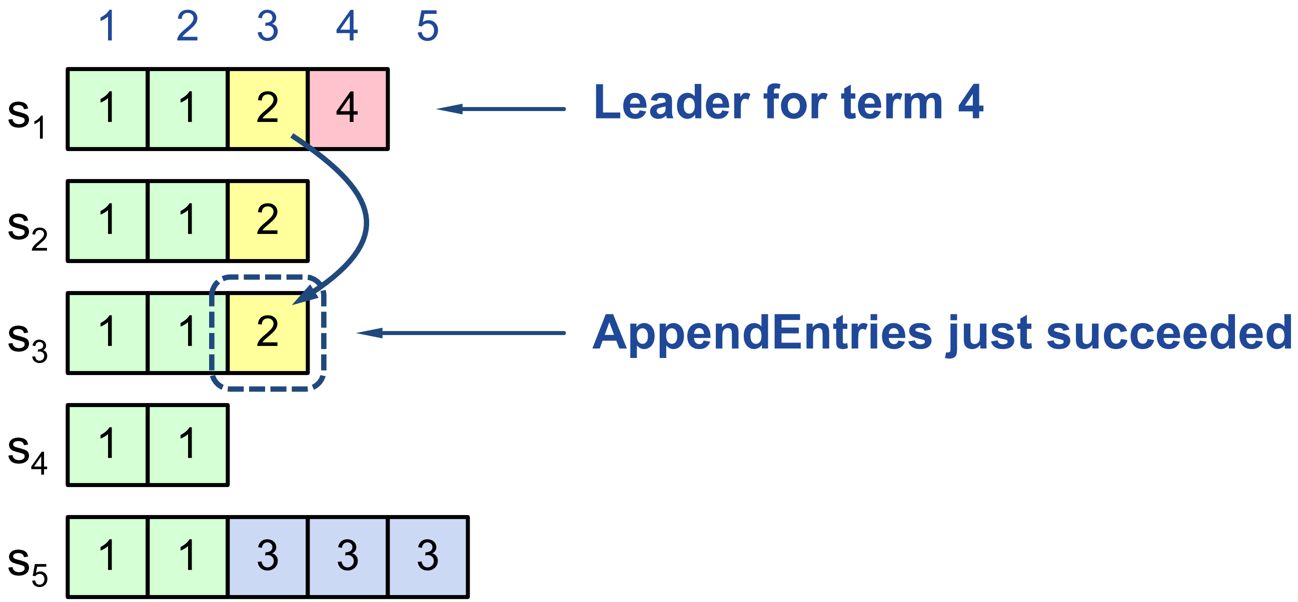
AppendEntries just succeeded

Can’t be elected as leader for term 3

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

- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term



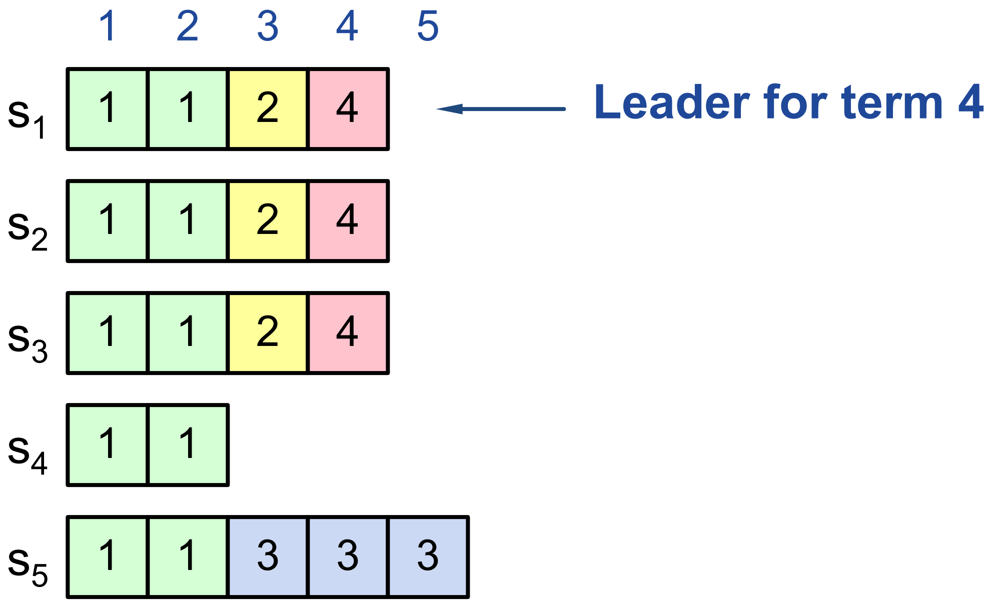
	1	2	3	4	5
s_1	1	1	2	4	
s_2	1	1	2		
s_3	1	1	2		
s_4	1	1			
s_5	1	1	3	3	3

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
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3

New Commitment Rules



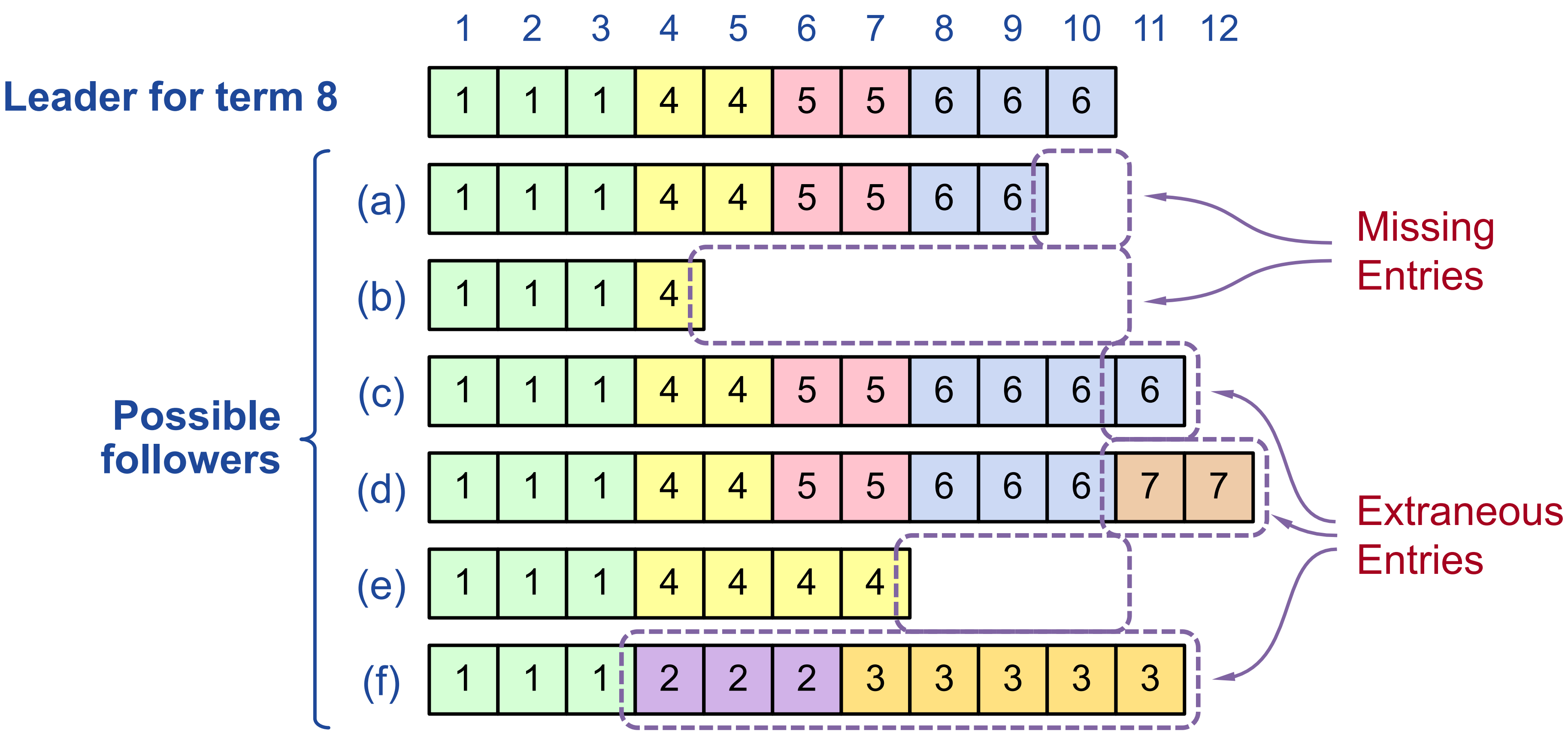
	1	2	3	4	5
s_1	1	1	2	4	
s_2	1	1	2	4	
s_3	1	1	2	4	
s_4	1	1			
s_5	1	1	3	3	3

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

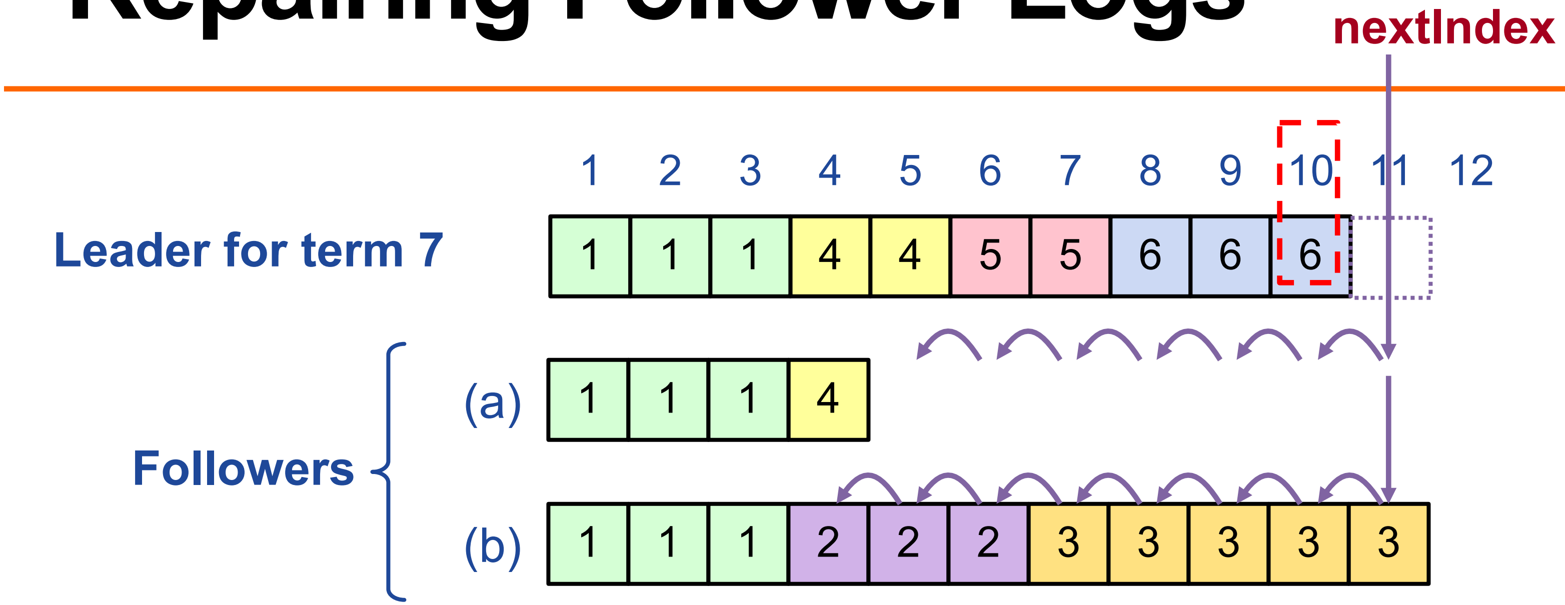
Challenge: Log Inconsistencies



	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	

Leader changes can result in log inconsistencies

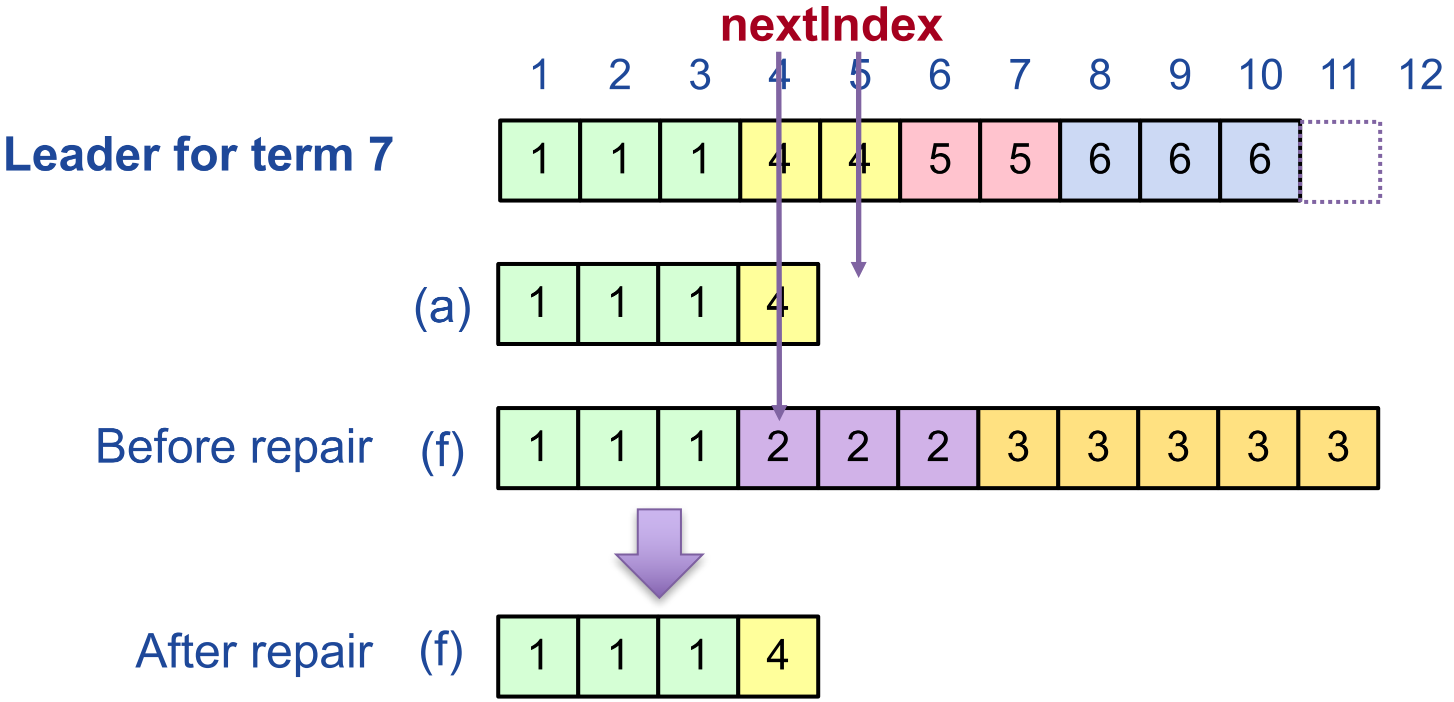
Repairing Follower Logs



		1	2	3	4	5	6	7	8	9	10	11	12
Leader for term 7		1	1	1	4	4	5	5	6	6	6		
Followers	(a)	1	1	1	4								
	(b)	1	1	1	2	2	2	3	3	3	3	3	

- **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



nextIndex

		1	2	3	4	5	6	7	8	9	10	11	12
Leader for term 7		1	1	1	4	4	5	5	6	6	6		
	(a)	1	1	1	4								
Before repair	(f)	1	1	1	2	2	2	3	3	3	3	3	
After repair	(f)	1	1	1	4								

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

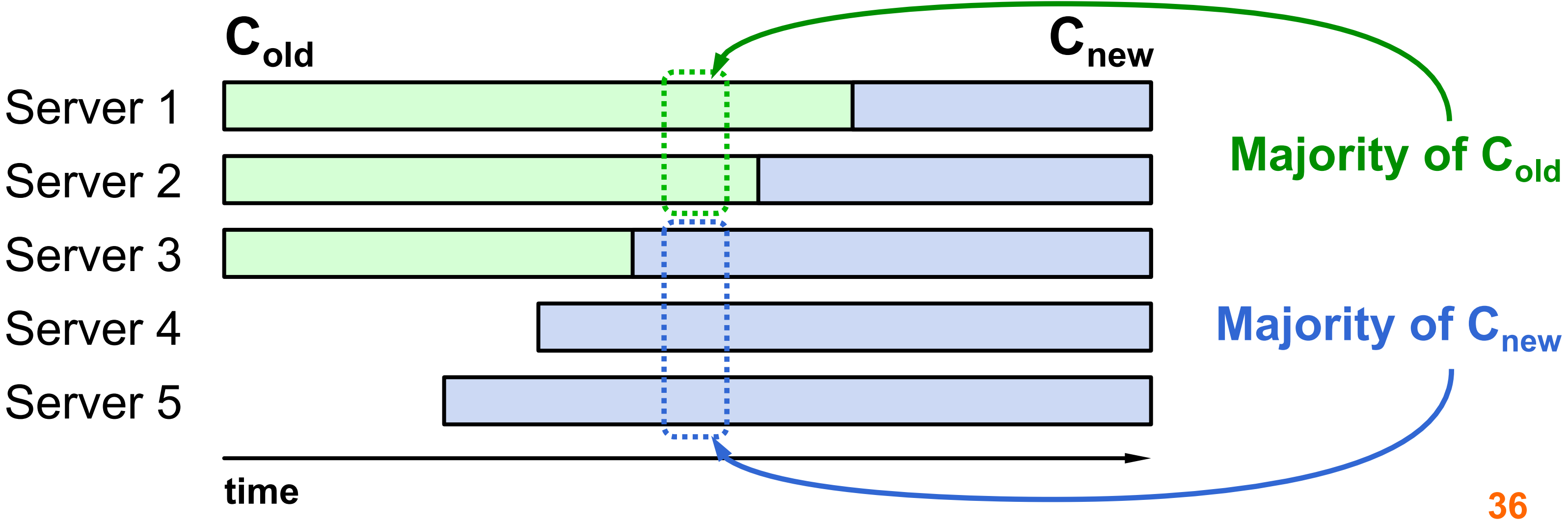
Reconfiguration

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

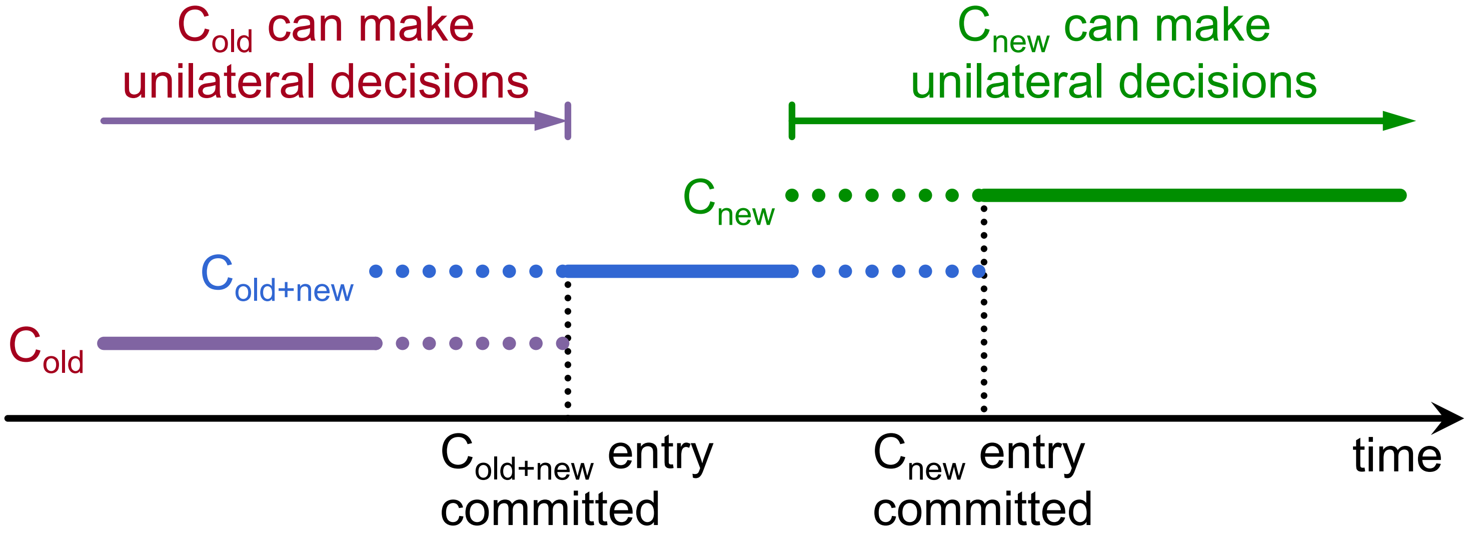


2-Phase Approach via Joint Consensus

- 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)

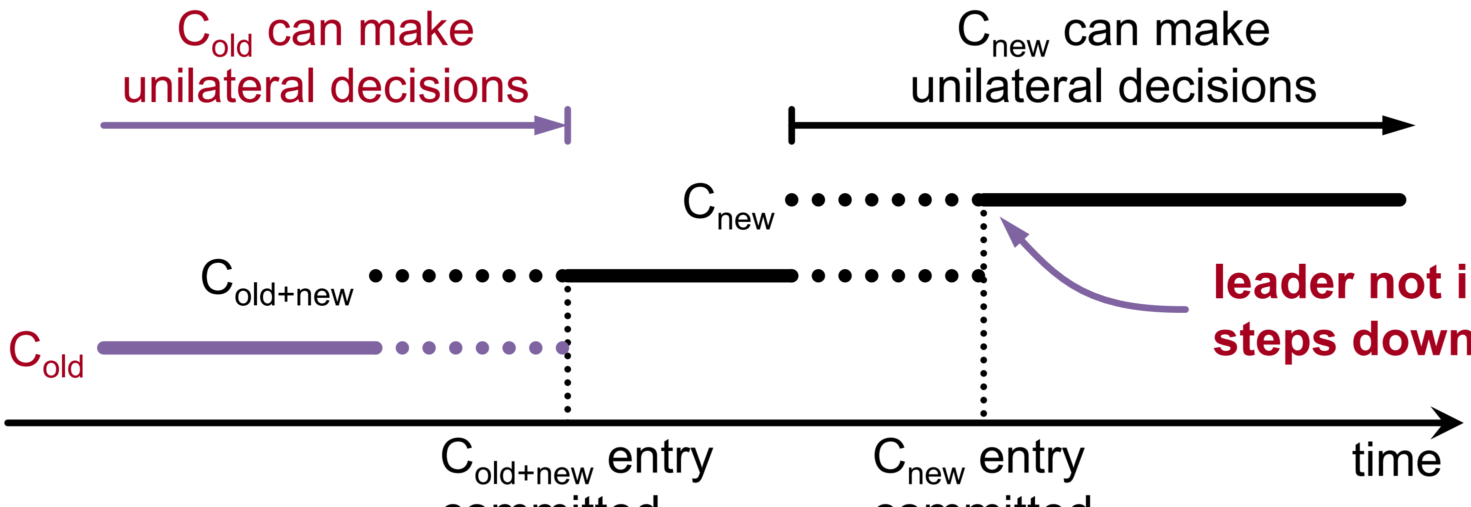
- Once joint consensus is committed, begin replicating log entry for final configuration


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

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


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

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

Wednesday lecture

Byzantine Fault Tolerance

Replicated State Machines
with arbitrary failures