FAULT TOLERANCE AND CONSENSUS WITH RAFT

George Porter
Feb 10, 2022

UC San Diego

ATTRIBUTION

- These slides incorporate material from:
  - Diego Ongaro and John Ousterhout, Stanford University

UC San Diego

REQUIRED READING

In Search of an Understandable Consensus Algorithm (Extended Version) by Diego Ongaro and John Ousterhout (https://raft.github.io/raft.pdf).

- Section 1, 2, 4, 5, 8, and 11 are required reading
- Sections 3, 6, 7, 9, 10, and 12 are optional and not necessary for your project
- You will **not** be implementing log compaction or membership changes!

To study for this topic, please refer to the paper. Consensus protocols are very subtle and studying these slides and/or rewatching the lecture will ***NOT*** be sufficient for obtaining a deep understanding of the RAFT protocol.

OUTLINE

1. Quorums
2. Fault tolerance
    1. Formulate system logic as a state machine
    2. Election to choose the transaction coordinator (TC)
    3. TC performs 2PC to replicate operations to multiple backup state machines
    4. Mechanism to “clean up” system when TC fails

ROADMAP ON APPROACHES TO FAULT TOLERANCE

	# servers that can fail before data is lost	Accepts updates and serves clients during failures	# servers that need to be operational to accept updates and serve clients	Transaction coordinator can fail
Single replica	0	No	1	N/A
N replicas w/ 2-PC	N-1	No	N	No
N replicas w/ (Nr,Nw) quorum	N_w - 1	Yes	N_w	No
N replicas w/ RAFT algorithm	N-1	Yes	N/2 + 1 (N is odd)	Yes

FORMULATE SYSTEM AS A STATE MACHINE

4. ***General Science:*** Two courses chosen from PHYS 2A, PHYS 2B, PHYS 4A, PHYS 4B, CHEM 6A or CHEM 6AH, CHEM 6B or CHEM 6BH, BILD 1, BILD 2, BILD 3, and BICD 100 (8 units)
5. ***Probability and Statistics:*** MATH 183 or ECE 109 or ECON 120A or CSE 103 (4 units)

2. Upper-Division Requirements

Students must complete 72 upper-division units: 44 units of [illegible] courses and 28 units of cluster and elective courses.

1. ***Core Courses:***
   - Data structures and programming: CSE 1[illegible]
   - Algorithms/theory: CSE 101 and CSE 105
   - Software engineering: CSE 110
   - Hardware/architecture: CSE 140 and CSE 141, along with CSE 140L and CSE141L
   - Systems/networks: CSE 120 or CSE 123 or CSE 124
   - PL/databases: CSE 130 or CSE 132A
   - Security/cryptography: CSE 107 or CSE 127
   - Learning/vision/graphics: CSE 150 or CSE 151 or CSE 152 or CSE 153 or CSE 158 or CSE 167

   Students are expected to complete the majority of these courses by the end of their junior year.

REPLICATED DETERMINISTIC FINITE STATE MACHINES

PLAYER 1

1. GO NORTH
2. GO NORTH
3. GO EAST
4. GET SWORD
5. OPEN DOOR
6. GO SOUTH
7. FIGHT DRAGON
8. GET LAMP

PLAYER 2

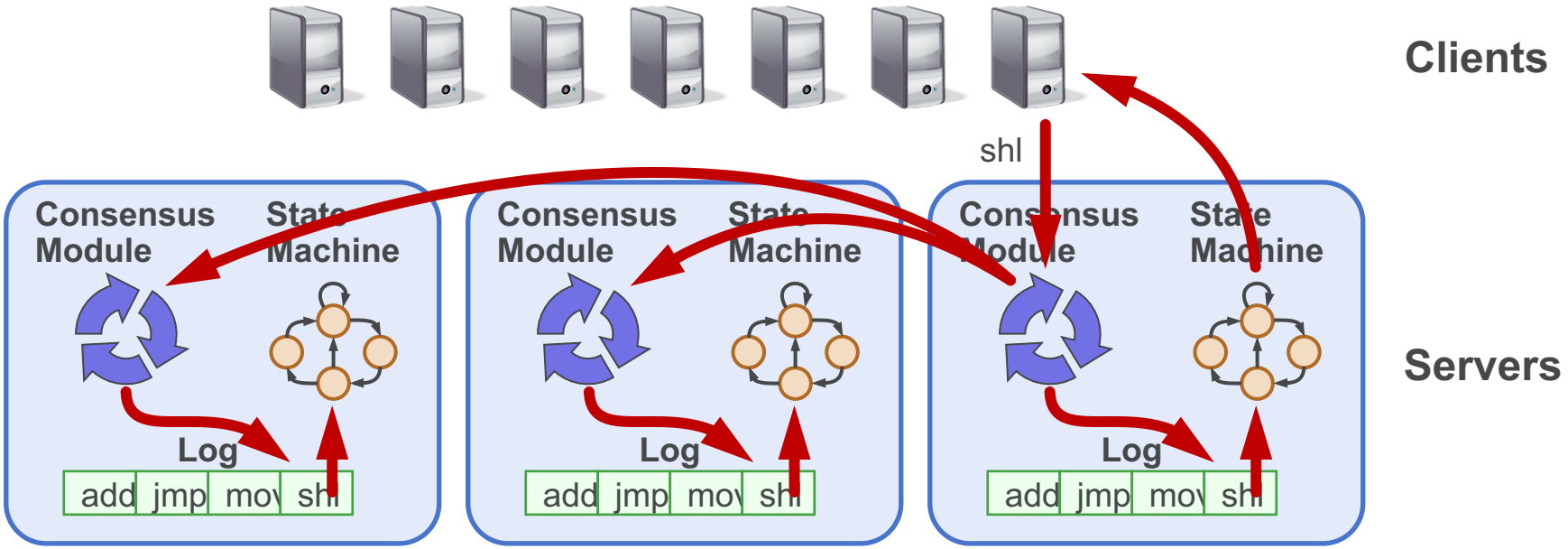
1. GO NORTH
2. GO NORTH
3. GO EAST
4. GET SWORD
5. OPEN DOOR
6. GO SOUTH
7. FIGHT DRAGON
8. GET LAMP

Both players in same state after same invocation of commands

STATE MACHINE REPLICATION

- Each machine starts in the same initial state
- Executes the same requests (deterministic)
- Requires consensus to execute in same order
  - (GET SWORD; FIGHT DRAGON) has a very different outcome from (FIGHT DRAGON; GET SWORD)
- Produces the same output

Goal: Replicated Log

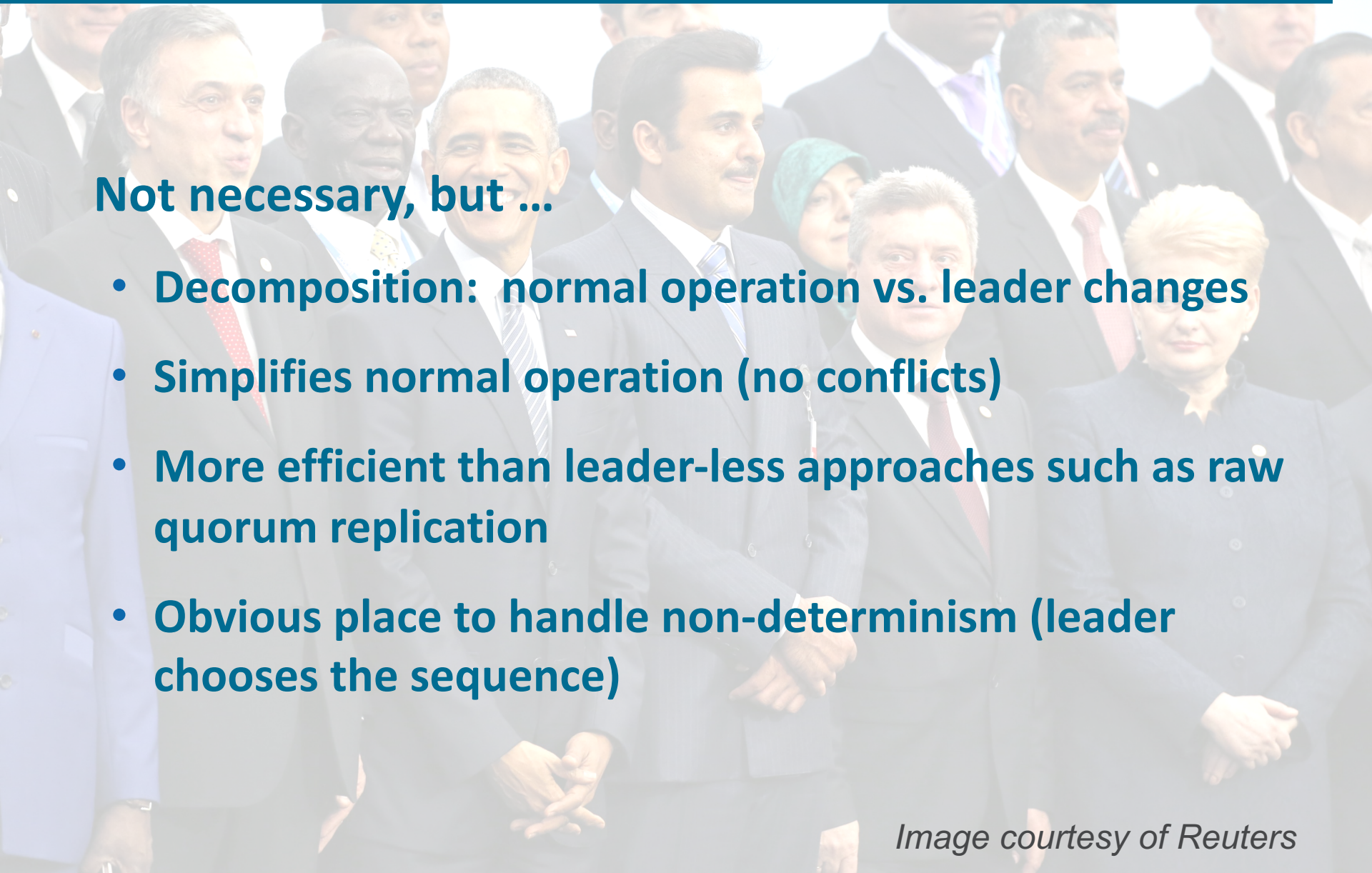


- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

OUTLINE

1. Quorums
2. Fault tolerance via RAFT
    1. Formulate system logic as a state machine
    2. Election to choose the transaction coordinator (TC)
    3. TC performs 2PC to replicate operations to multiple backup state machines
    4. Mechanism to “clean up” system when TC fails

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 such as raw quorum replication**
- **Obvious place to handle non-determinism (leader chooses the sequence)**

Image courtesy of Reuters

SERVER STATES

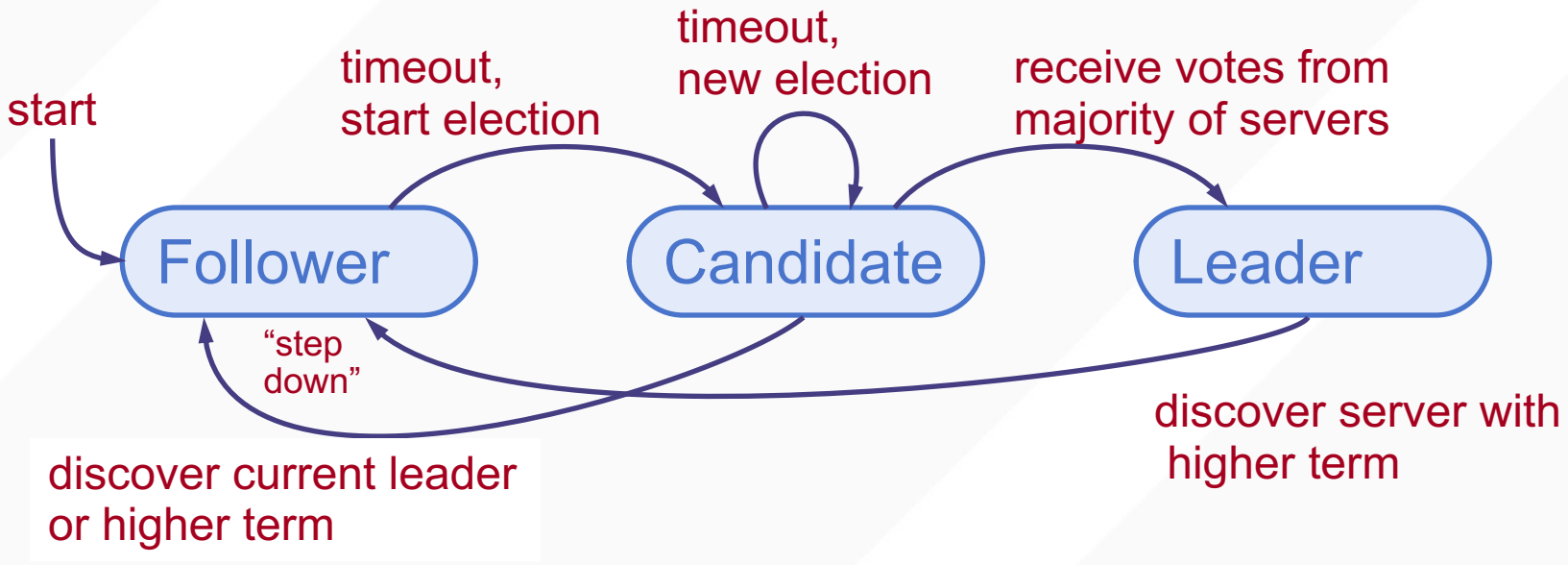
- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
  - Follower: completely passive
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

OPERATIONS

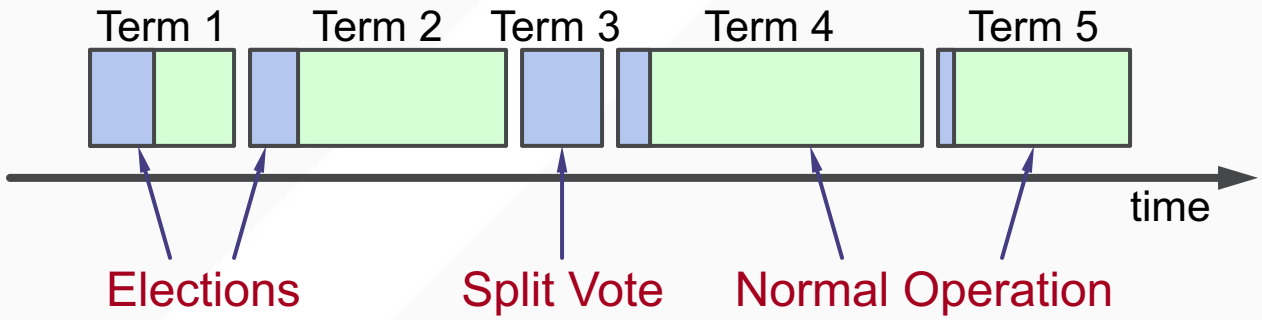
- AppendEntries()
  - The TC (leader) uses this to “push” new operations to the replicated state machines
  - Also used by the TC to tell the other nodes it is the TC/leader
- RequestVote()
  - Used when the system starts up to select a leader
  - Used when the leader fails to elect a new leader
  - Used when the leader is unreachable due to a network partition to elect a new 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)



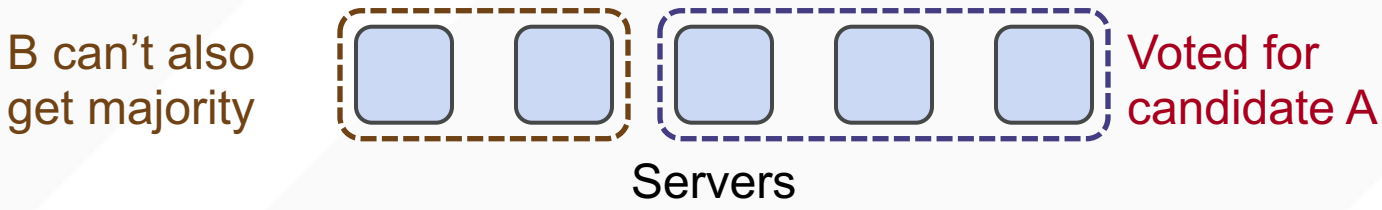
- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains current term value
- Key role of terms: identify obsolete information

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

ELECTION PROPERTIES

- 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

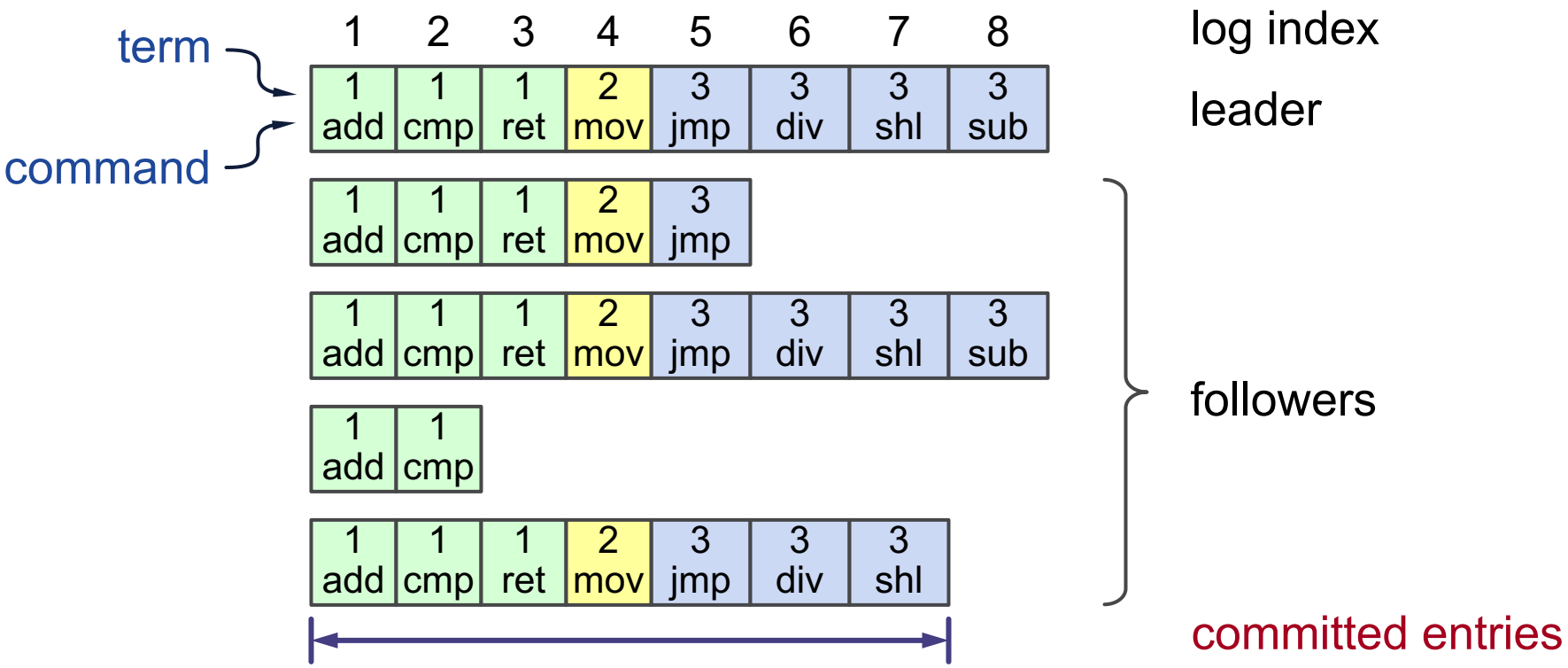
WEB SIMULATOR/DEMO

https://raft.github.io/raftscope/index.html

RAFT OVERVIEW

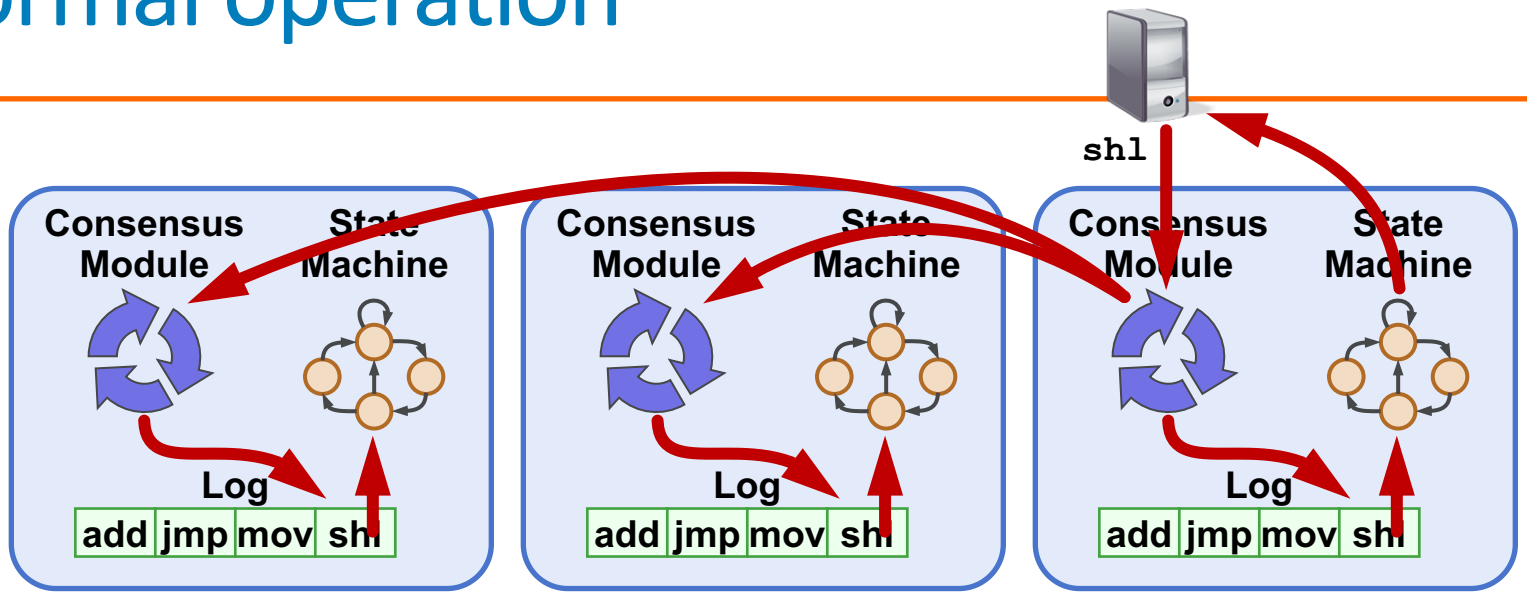
1. ~~Leader election~~
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. ~~Reconfiguration~~

Log Structure



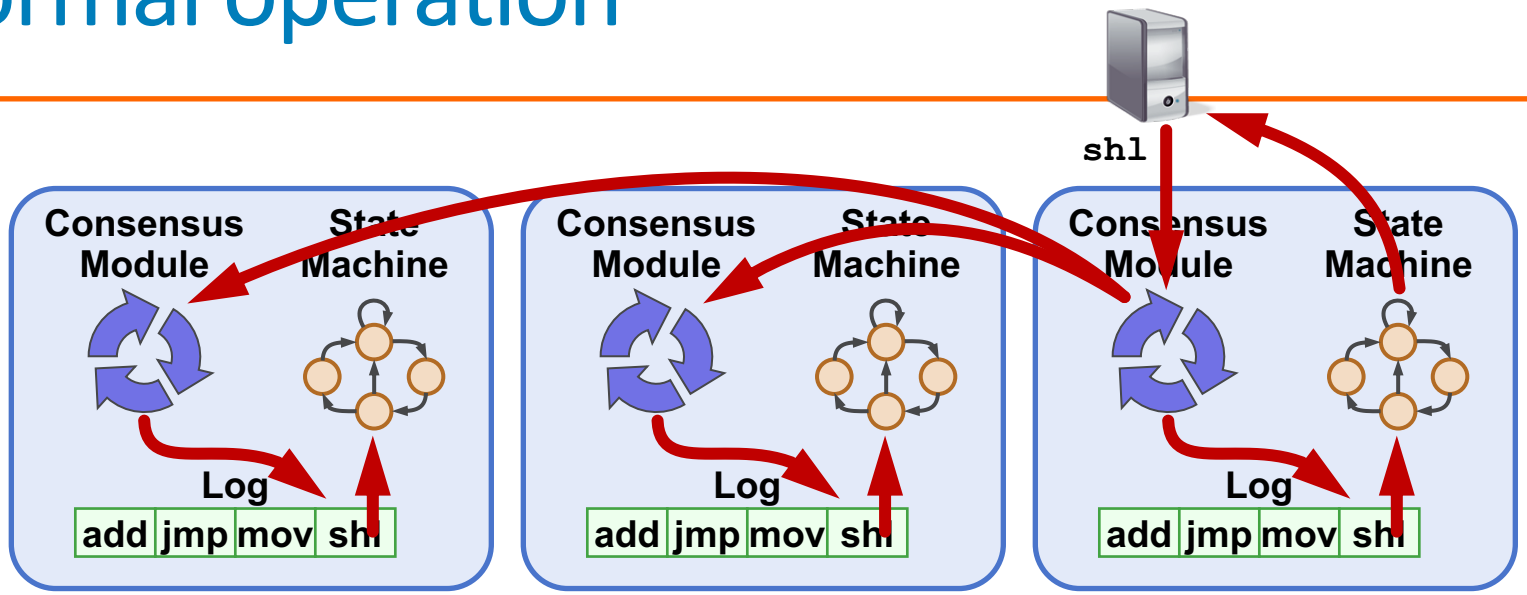
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

Normal operation



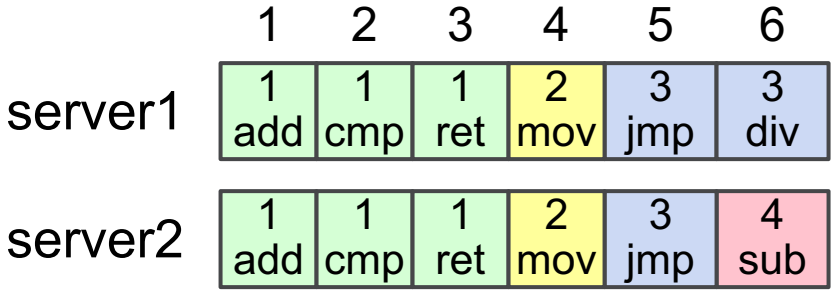
- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, sends result to client
  - Leader piggybacks commitment to followers in later AppendEntries
  - Followers pass committed commands to their state machines

Normal operation



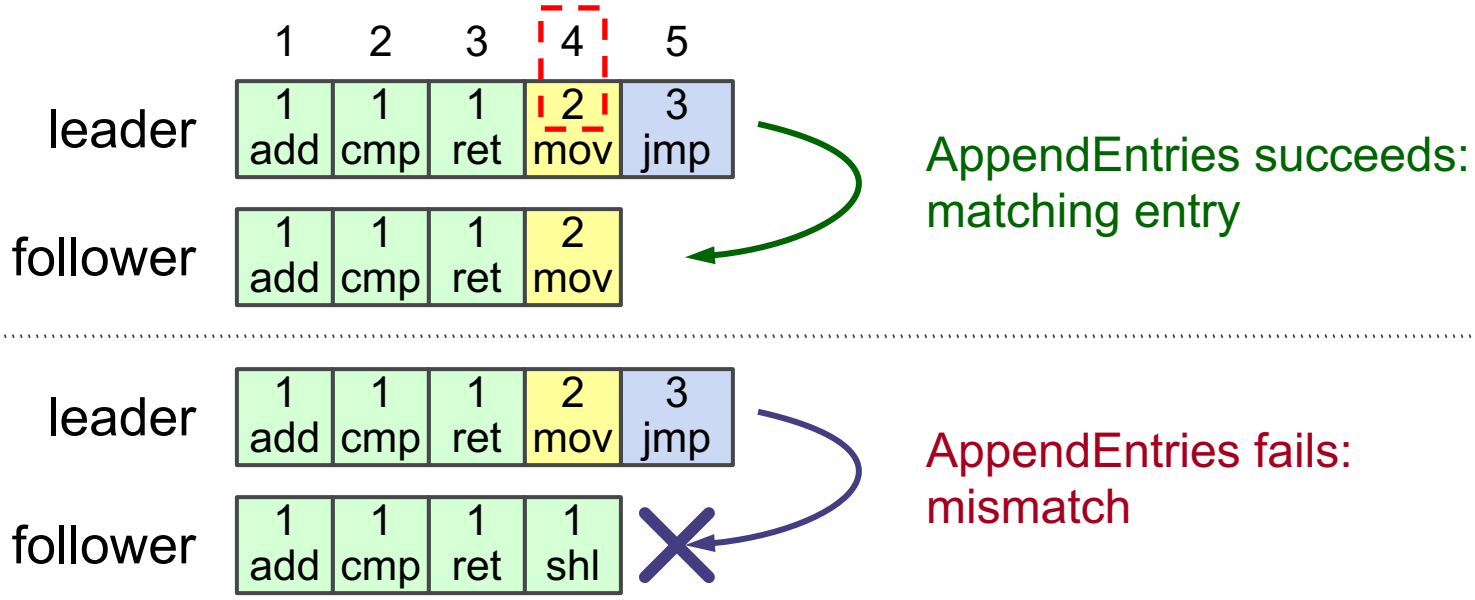
- Crashed / slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

Log Operation: Highly Coherent



- 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

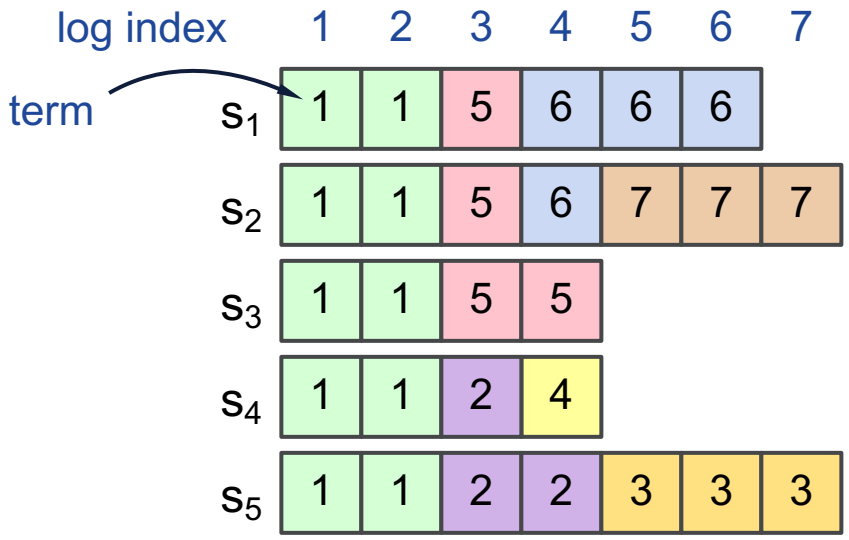


- AppendEntries has <index,term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an induction step, ensures coherency

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



Safety Requirement

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

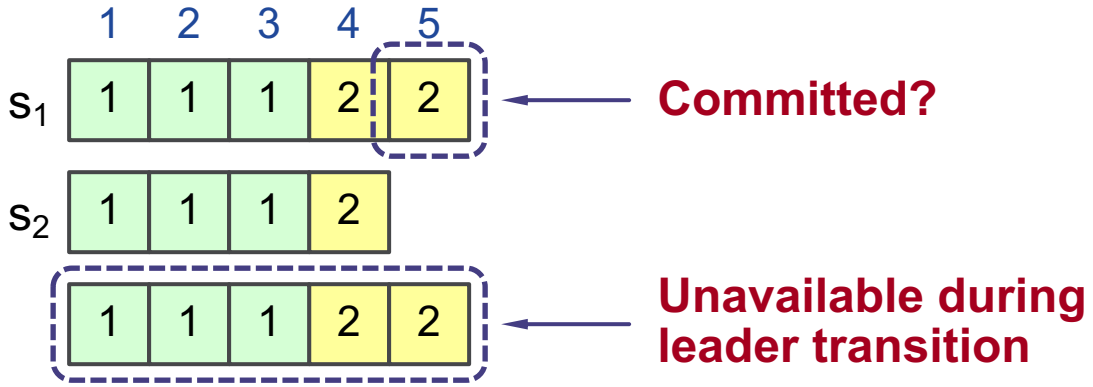
- Raft safety property: If leader has decided log entry is committed, entry will be present in logs of all future leaders

- 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

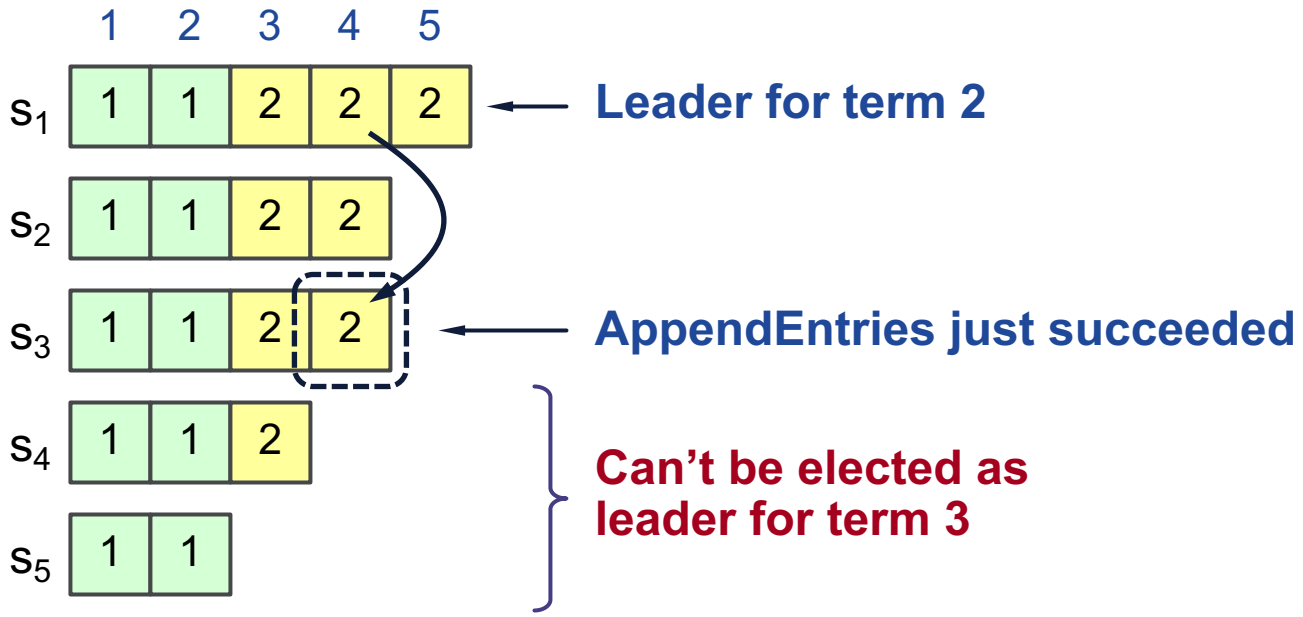
Picking the Best Leader

Can't tell which entries committed!



- 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

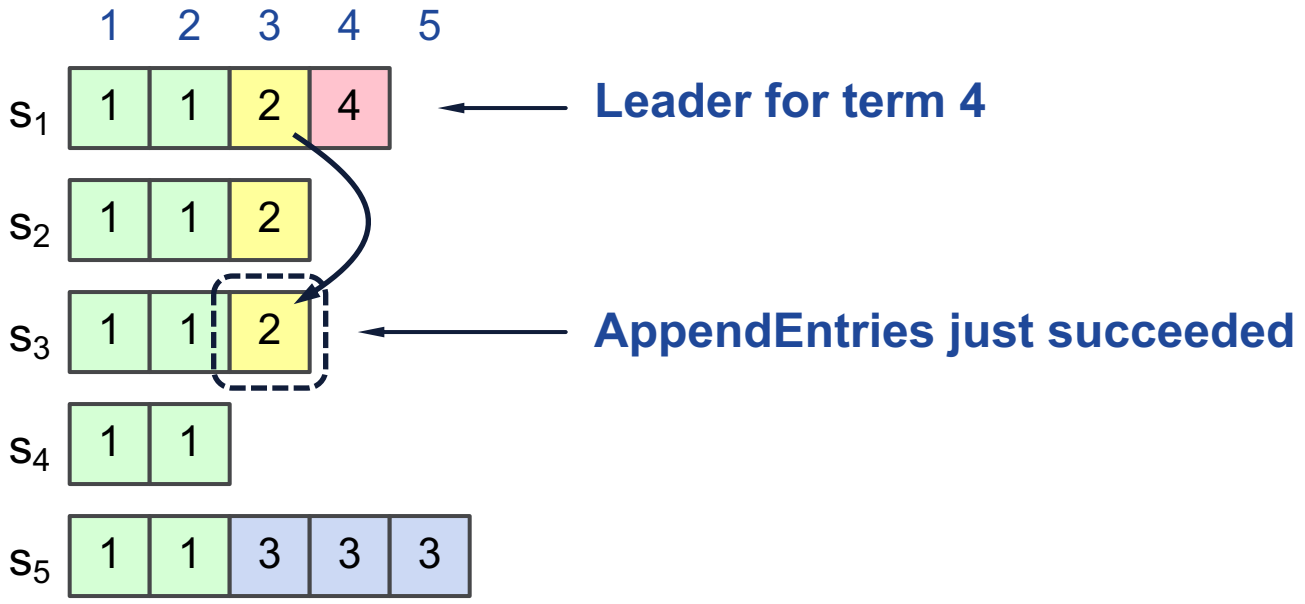
Committing Entry from Current Term



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

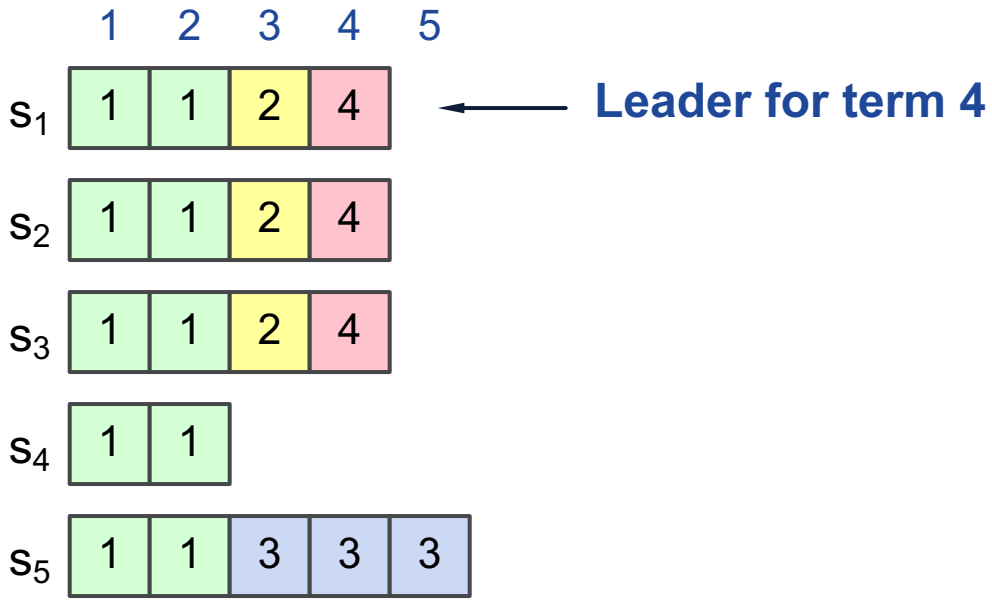
- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term



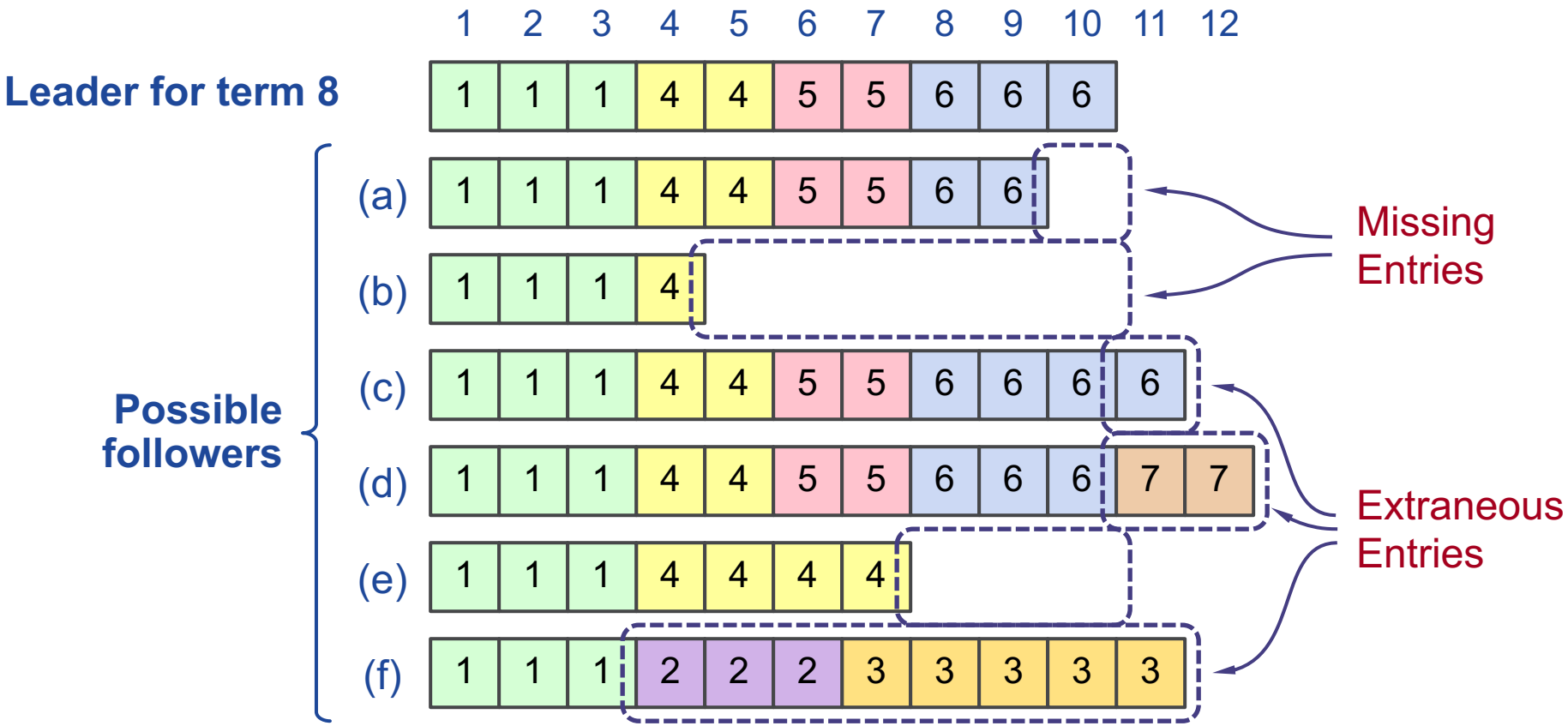
- Case #2: Leader trying to finish committing entry from earlier
- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5 (how?)
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3

New Commitment Rules



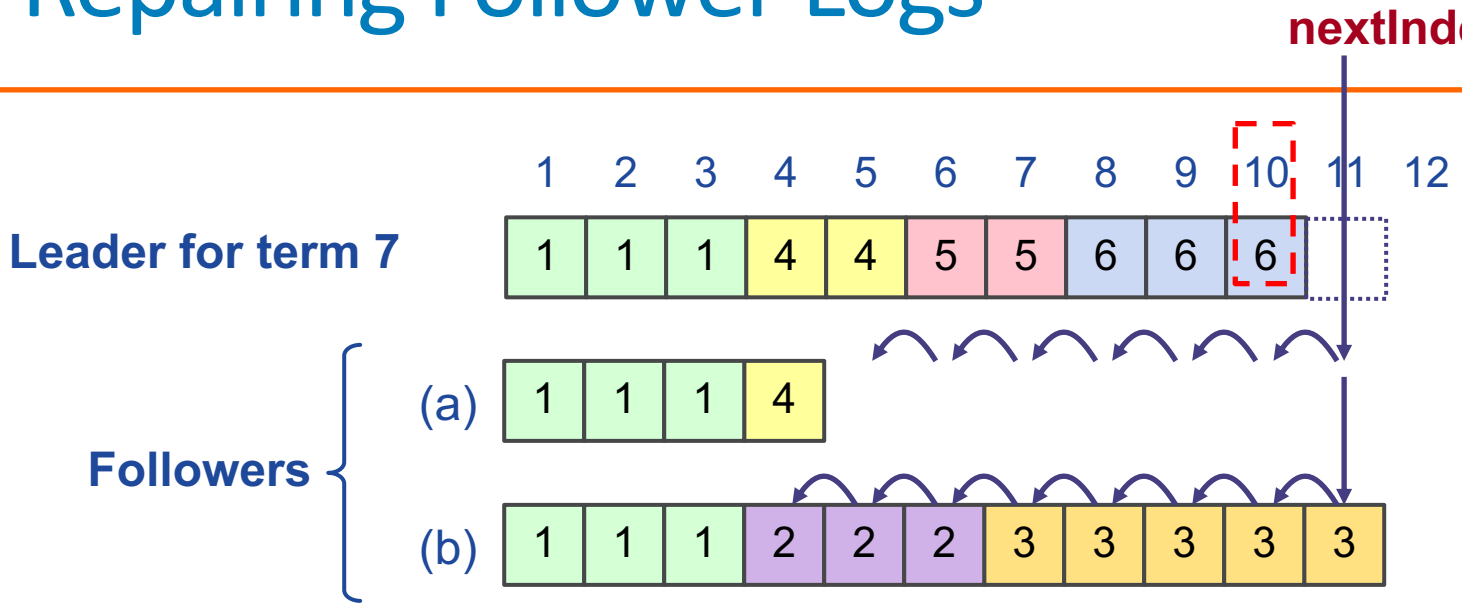
- For leader to decide entry is committed:
  1. Entry stored on a majority
  2. ≥ 1 new entry from leader's term also on majority

Challenge: Log Inconsistencies



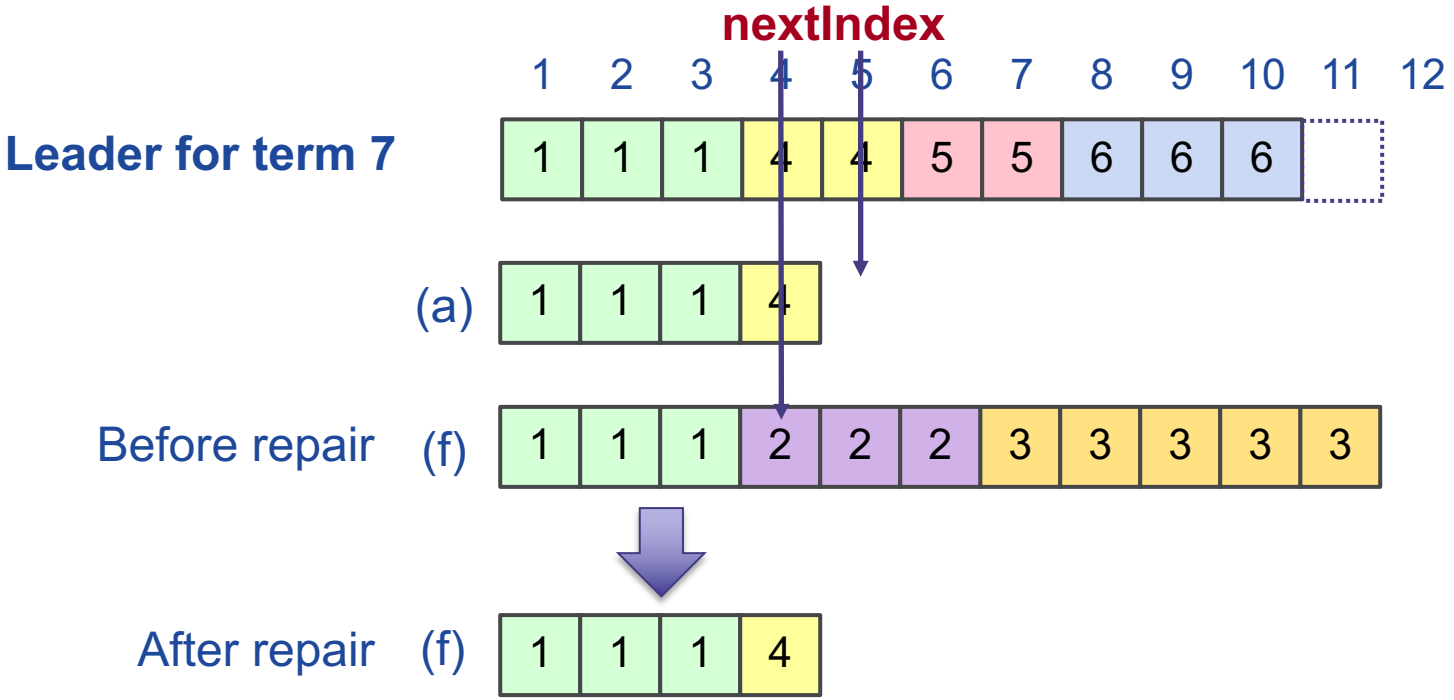
Leader changes can result in log inconsistencies

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 + 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

WEB SIMULATOR/DEMO

https://raft.github.io/raftscope/index.html

UC San Diego