UNIVERSITY of VIRGINIA

CS4740
CLOUD COMPUTING

Consensus (Contd.)

Prof. Chang Lou, UVA CS, Fall 2026

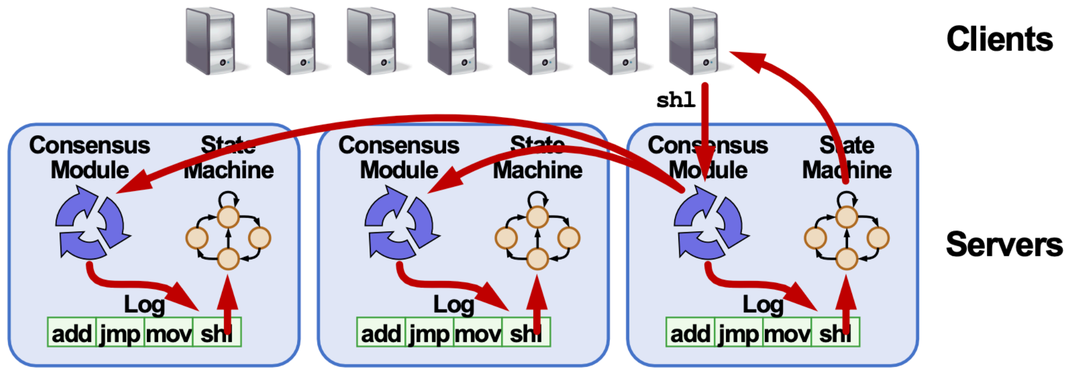
CONTEXT

— Today we continue talking about Raft, but in more details.

GAME: CONSENSUS (REVENGE)!

- Each student votes on an integer between 1 - 100.
  - Can only vote once, repeated votes become invalid.
- **Basic**: everyone who votes gets 1 extra pt in the final grade!
- **Extra**: if a valid super-quorum (>70% total votes) is reached, everyone in the super-quorum gets another extra pt.
- **Extra2**: n **agents** have joined the game (they remain anonymous even after game).
  - Their goal is to create a split and fail the super-quorum. They have their own bonus rule.
  - They do not know each other and they cannot communicate with Chang.
  - Chang will still void a number, but agents cannot leak data to Chang.
- You have 5 min to discuss a strategy.
- Submit your votes to https://forms.gle/nLeDuuw3ff3jWZ5U7

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
  - when to commit?
  - how to fix inconsistent logs?
- 4. Client interactions

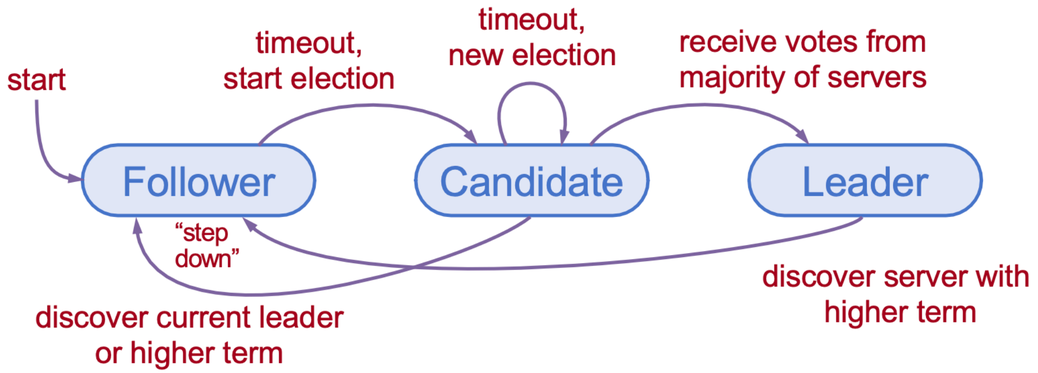
SERVER STATES

- 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
  - **Exercise**: how do states transit to others?

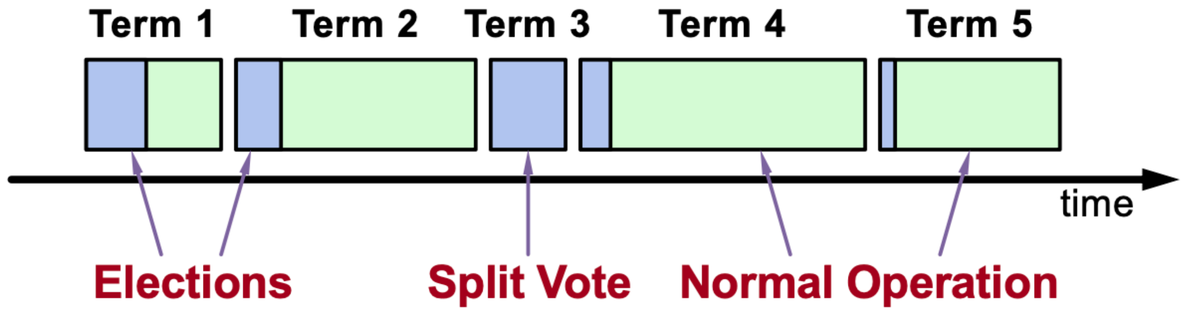


STATE TRANSITION

- Servers start as followers
- Leaders send heartbeats (empty AppendEntries RPCs) to maintain authority over followers
- 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

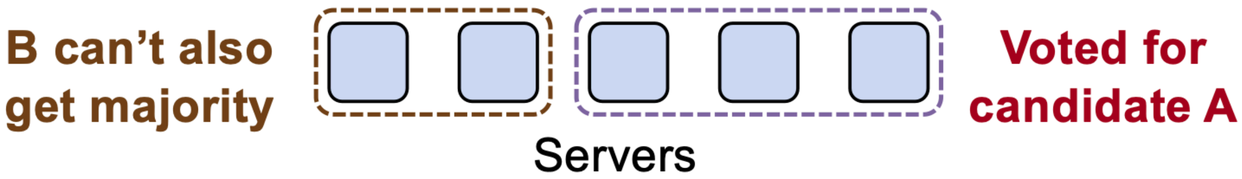
- Start election:
  - Increment current term, change to candidate state, vote for self
- Send Request Vote to all other servers, retry until either:
  - 1. Receive votes from majority of servers:
    - Become leader
    - Send AppendEntries heartbeats to all other servers
  - 2. Receive RPC from valid leader:
    - Return to follower state
  - 3. No-one win selection (election timeout elapses):
    - Increment term, start new election

TWO PROPERTIES

- **Safety**
  - only good things happen
- **Liveness**
  - good things will eventually happen

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



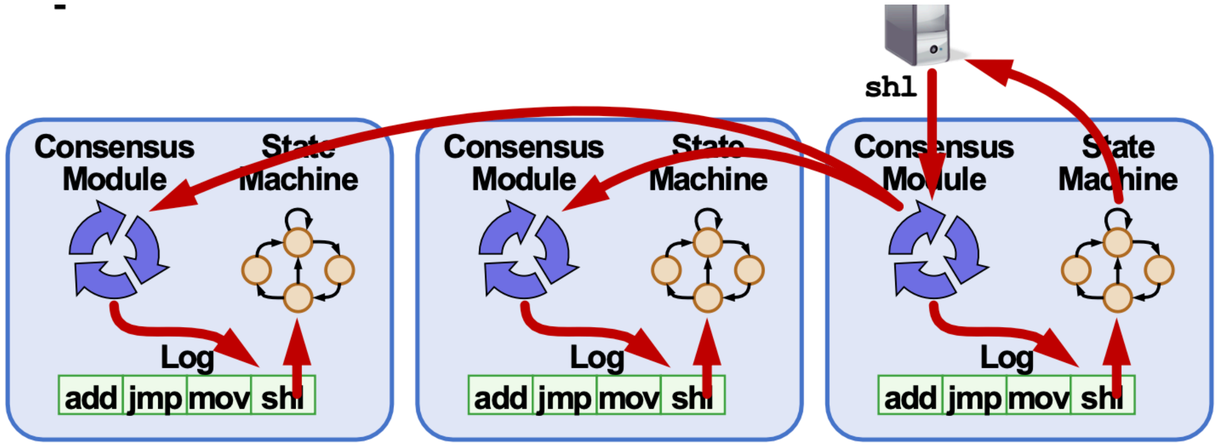
- Liveness: some candidate eventually wins
  - Each choose election timeouts randomly in [T, 2T]
  - One usually initiates and wins election before others start
  - Works well if T >> network RTT

LOG REPLICATION: RAFT W/ VISUALIZATION

https://thesecretlivesofdata.com/raft/#replication

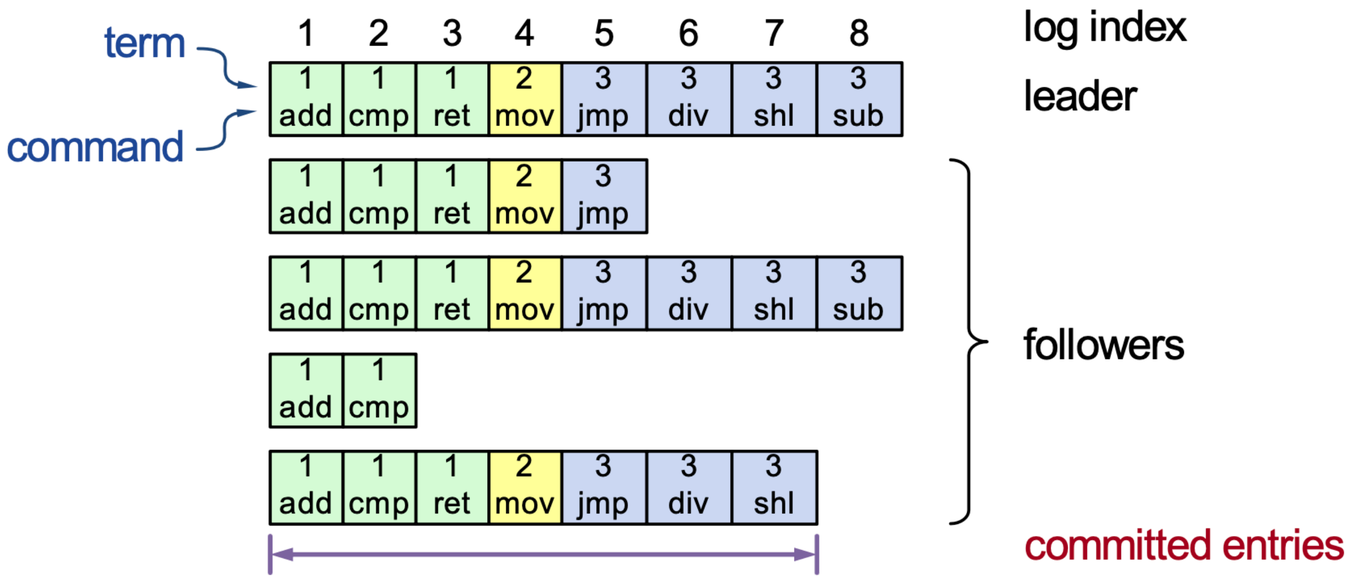
NORMAL OPERATION

-



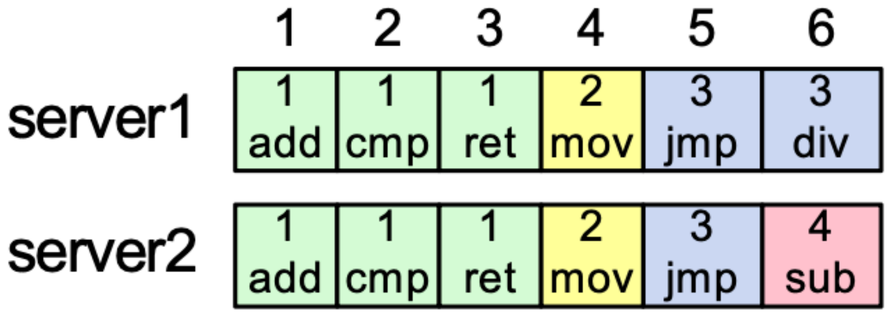
- Client sends command to leader
- Leader appends command to its log
  - **Leader Append-Only Property**: a leader never overwrites or deletes entries in its log; it only appends new entries.
- Leader sends AppendEntries RPCs to followers, once new entry committed:
  - Leader passes command to its state machine, sends result to client
  - Leader piggybacks commitment to followers in later AppendEntries
  - Followers pass committed commands to their state machines

LOG STRUCTURE



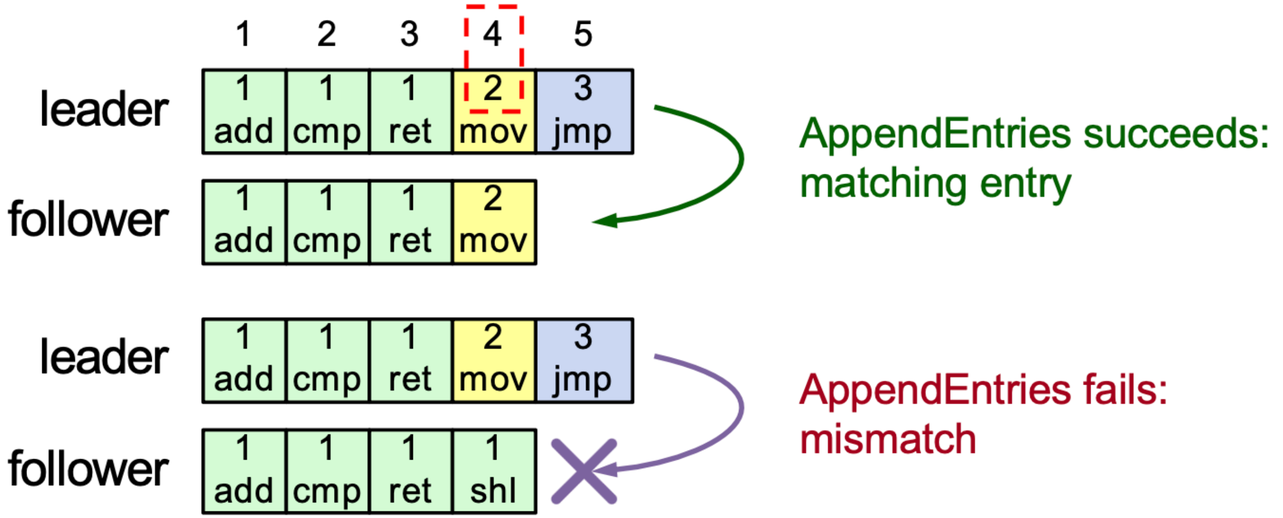
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed -> stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

LOG MATCHING PROPERTY



- If log entries on different server have same index and term:
  - Store the same command => one leader, one entry, one index, one term + log position never change
  - Logs are identical in all preceding entries => consistency check
- If given entry is committed, all preceding also committed

LOG OPERATION: CONSISTENCY CHECK



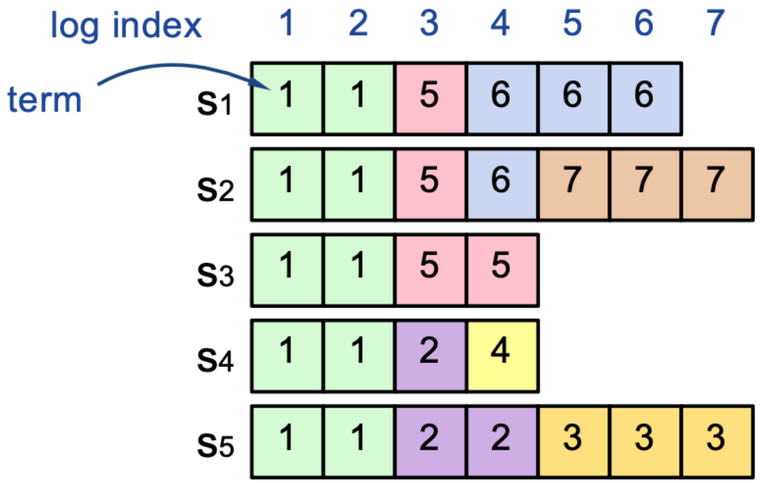
- AppendEntries has <index,term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an induction step, ensures coherency

RAFT VISUALIZATION

https://raft.github.io/

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 (?)

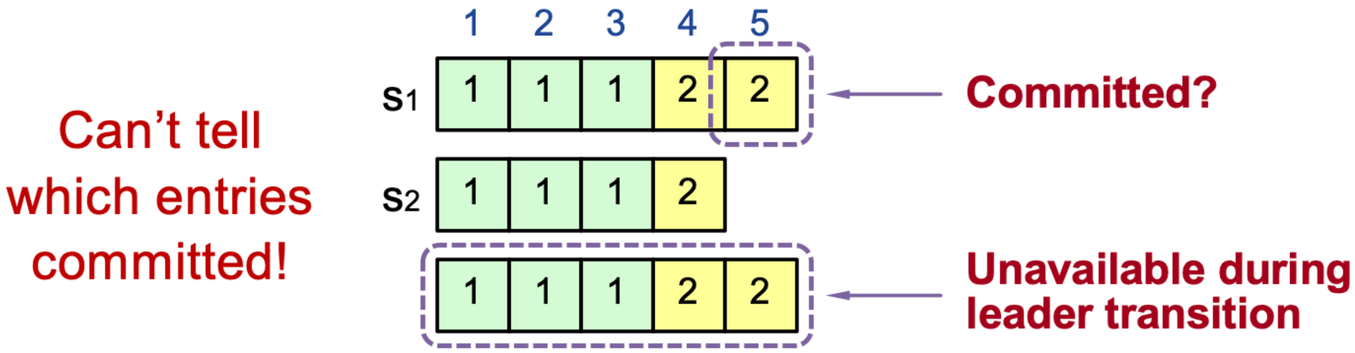


SAFETY REQUIREMENT

- Leader Completeness 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 over write entries in their logs
  - 2. Only entries in leader's log can be committed
  - 3. Entries must be committed before applying to state machine

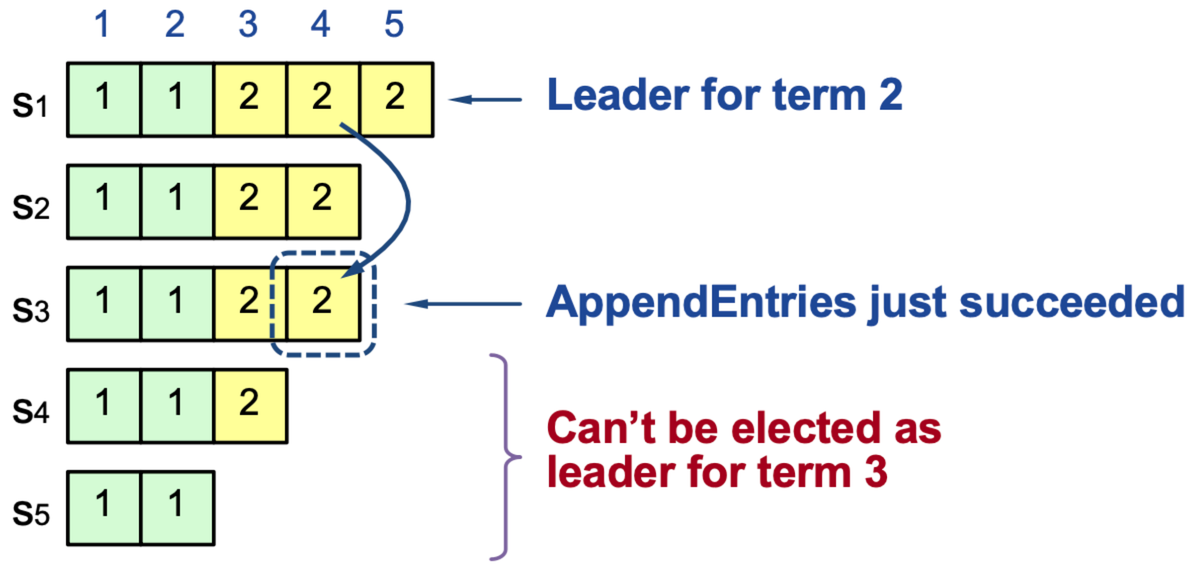


PICKING THE BEST LEADER



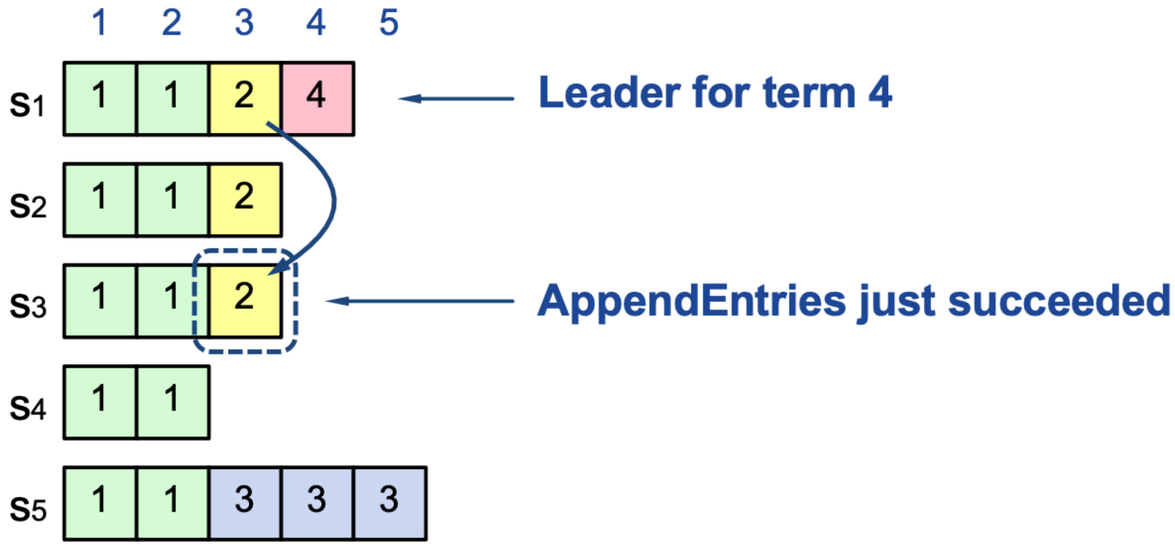
- Who should be the leader?
- Elect candidate contains all committed entries!
  - In RequestVote, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete": (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among 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
- What happened?
- Is entry 3 safely committed? (it's stored on a majority)

COMMITTING ENTRY FROM EARLIER TERM

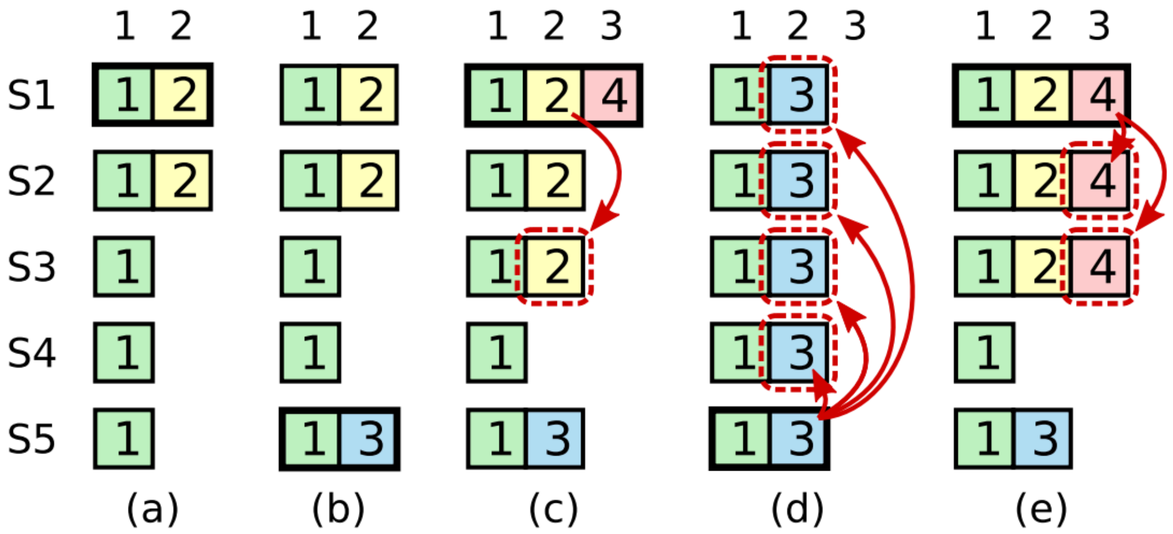
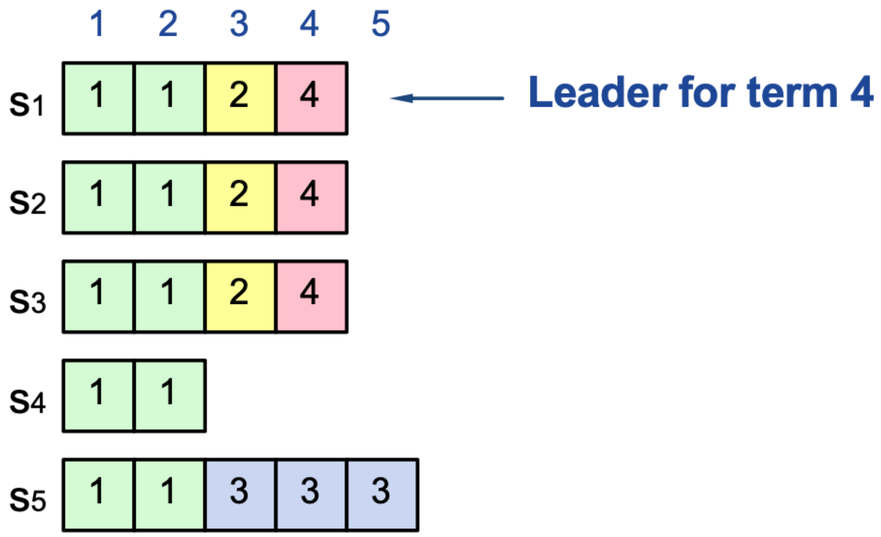


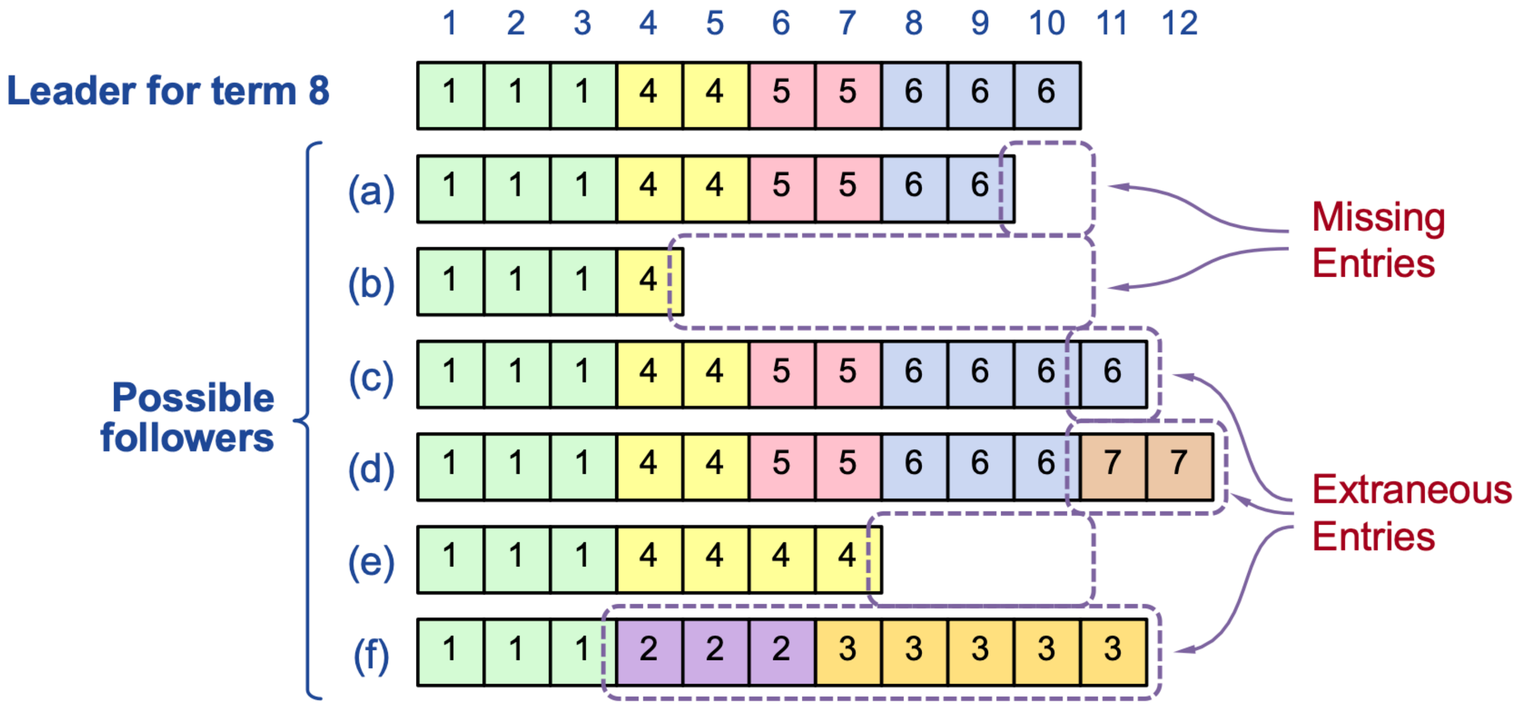
Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

NEW COMMITMENT RULES



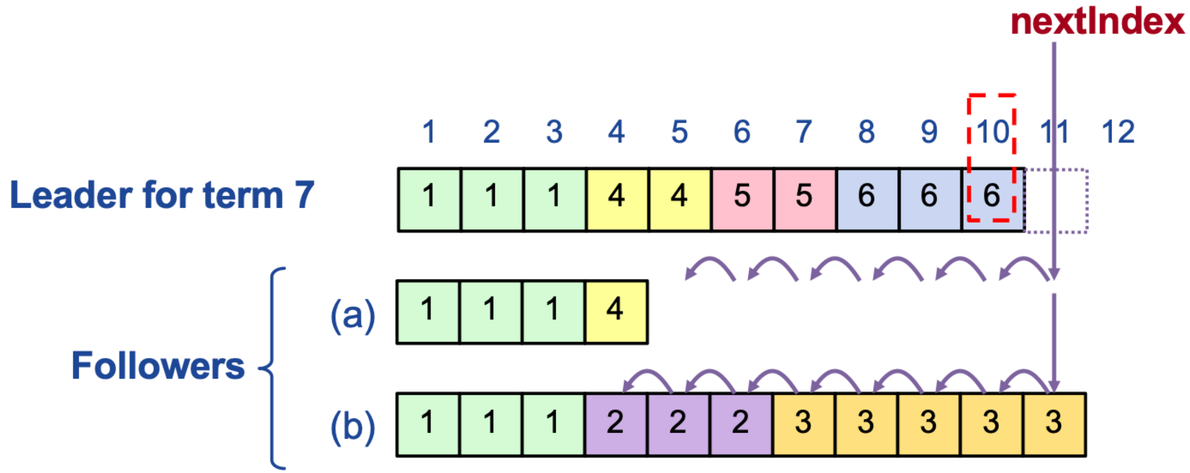
- Storing on a majority does not mean the entry is committed!
- For leader to decide entry is committed:
  - (1) Entry stored on a majority + (2) ≥1 new entry from leader's term also on majority
- Example; Once e4 committed, s5 cannot be elected leader for term 5, and e3 and e4 both safe

CHALLENGE: LOG INCONSISTENCIES



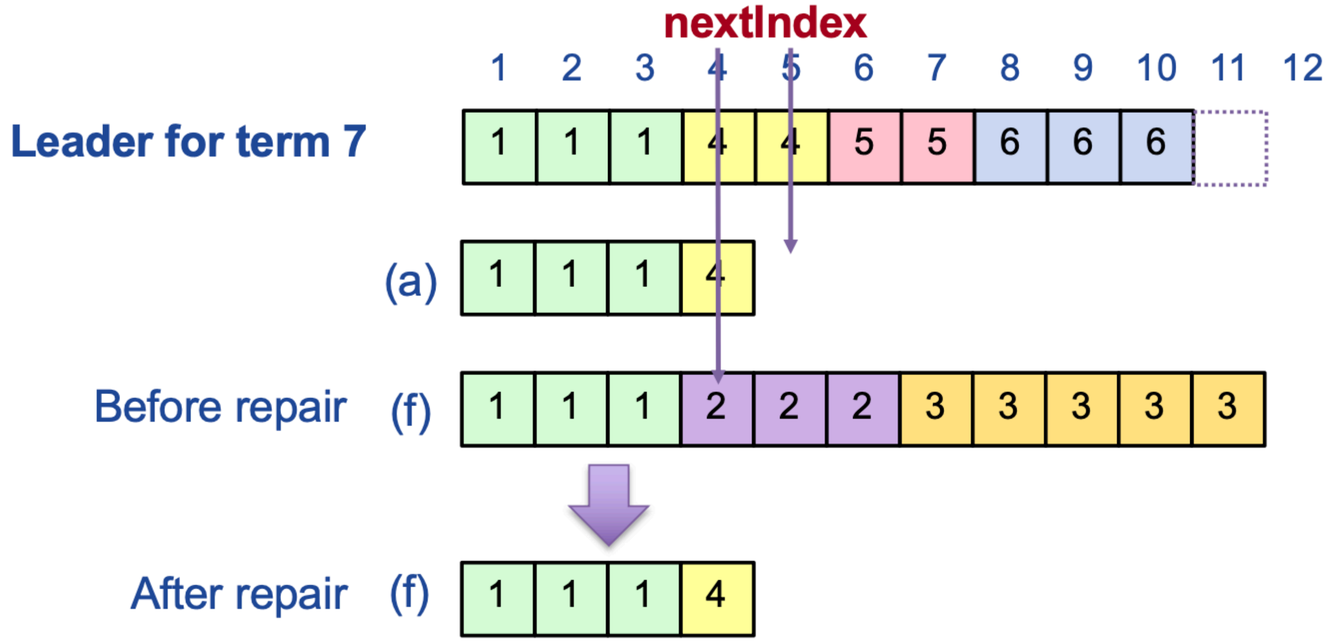
— Leader changes can result in log in consistencies

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



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

CLIENT PROTOCOL

- How to handle read-only operations?
  - Current leader responds without writing anything to log
  - Risk: Could return stale data if the leader has been replaced and doesn't know it.
- #1: Leader must know committed entries.
  - The Leader Completeness Property ensures a leader has all committed entries.
  - However, it may not know which are committed at term start.
  - Solution: Leader commits a no-op entry at the start of its term.
- #2: Leader must confirm it's still current.
  - To avoid serving stale reads after being deposed.
  - Solution: Leader exchanges heartbeats with a majority before processing read-only requests.

GAME: CONSENSUS RESULT

— https://docs.google.com/forms/d/1sZTHC4a7vUgK4l0Tuto5CXZeesCsXP9brl37B38NsBI/edit#responses

UNIVERSITY of VIRGINIA

TAKEAWAYS

- Node roles: Leader, Candidate, Follower
- Terms: to identify obsolete information
- Committed log: durable on the majority of nodes (+ new entry rule)
- Repair logs: Delete extraneous entries + Fill in missing entries
- Next class: **Midterm Review**.



ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.

THIS SLIDES INCLUDES CONTENTS FROM PROF. DAN PORTS' DISTRIBUTED SYSTEMS COURSE (UW)