Distributed Systems

CS425/ECE428

March 8 2023

Instructor: Radhika Mittal

Logistics

- HW2 is due today.

- Please checkout post #378 on CampusWire regarding your midterm exams.
    - Reserve a slot if you haven’t already.
    - Submit your Letters of Accommodations to CBTF, if required.
    - Syllabus: everything coverded in class upto and including Paxos.
    - Closed-book exam: cannot refer to any materials / cheat sheets.
    - CBTF will provide calculator and scratch paper.

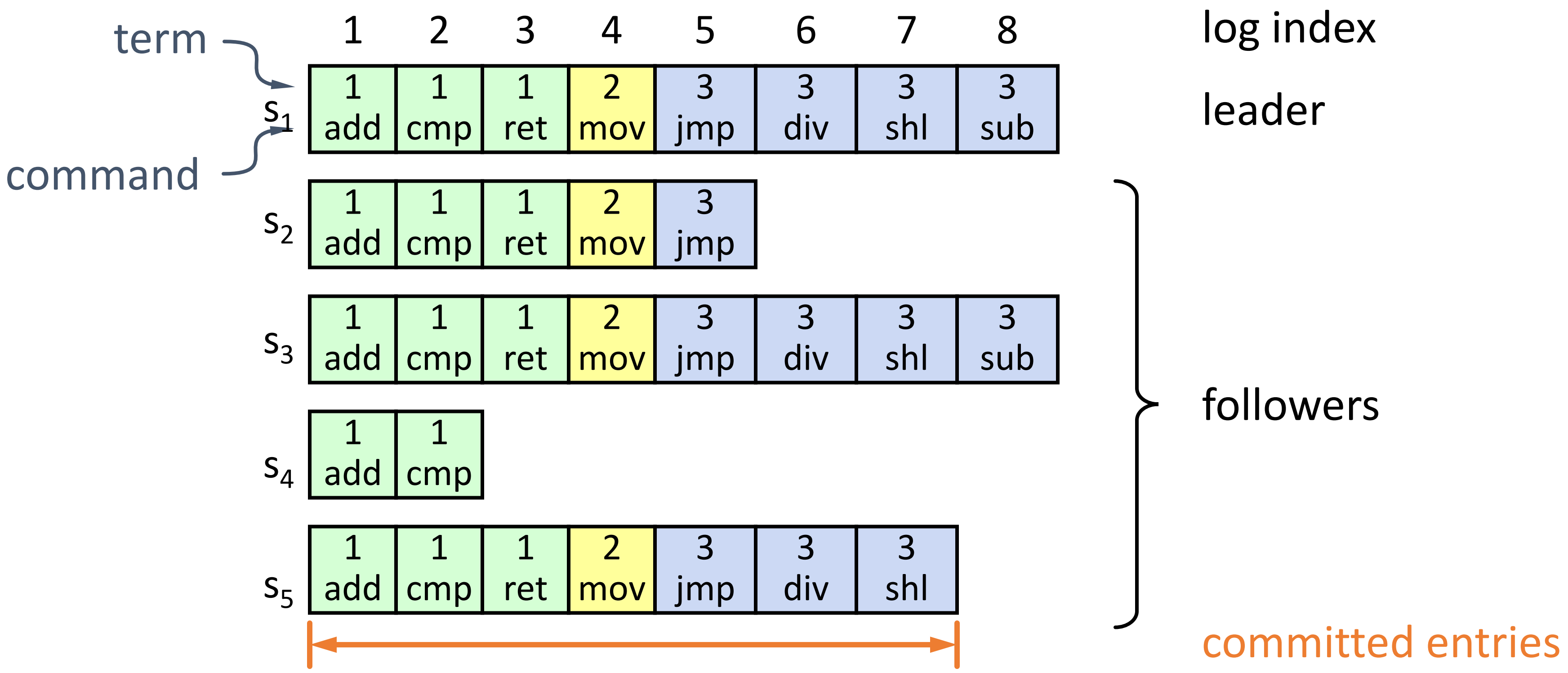
Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - Block-chains (distributed consensus)

Raft Recap

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
   - Time is divided into monotonically increasing "terms".
     - Each term starts with a leader election.
   - Only one leader can be per term.
2. Neutralizing old leaders
   - Use "terms" exchanged with RPCs to step down.
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

Log Structure


term
command
log index
leader
followers
committed entries
1 2 3 4 5 6 7 8
s1: 1 add, 1 cmp, 1 ret, 2 mov, 3 jmp, 3 div, 3 shl, 3 sub
s2: 1 add, 1 cmp, 1 ret, 2 mov, 3 jmp
s3: 1 add, 1 cmp, 1 ret, 2 mov, 3 jmp, 3 div, 3 shl, 3 sub
s4: 1 add, 1 cmp
s5: 1 add, 1 cmp, 1 ret, 2 mov, 3 jmp, 3 div, 3 shl

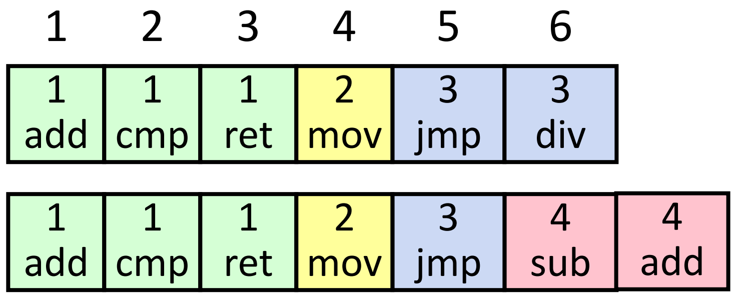
- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes
- Entry is committed by the leader when certain conditions are met*.
  - Durable, will eventually be executed by state machines
  - * we will get back to this.

Log Consistency

High level of coherency between logs:

Raft guarantees that:

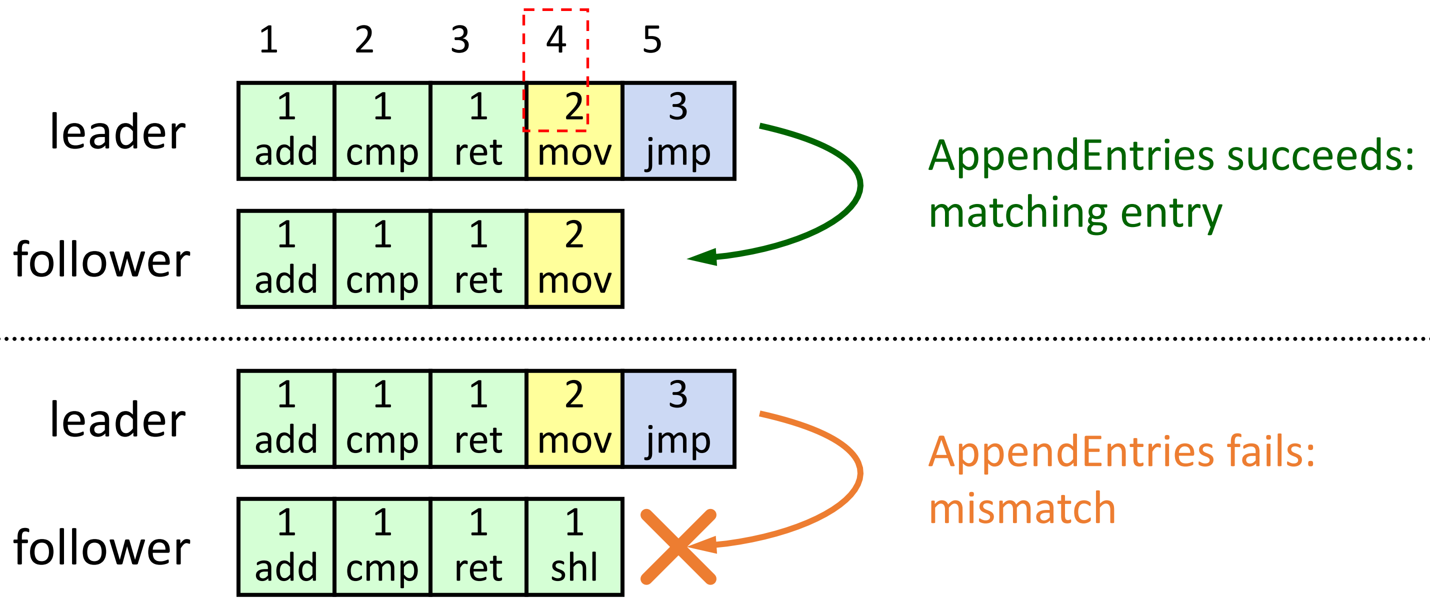
- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries



- If a given entry is committed, all preceding entries are also committed

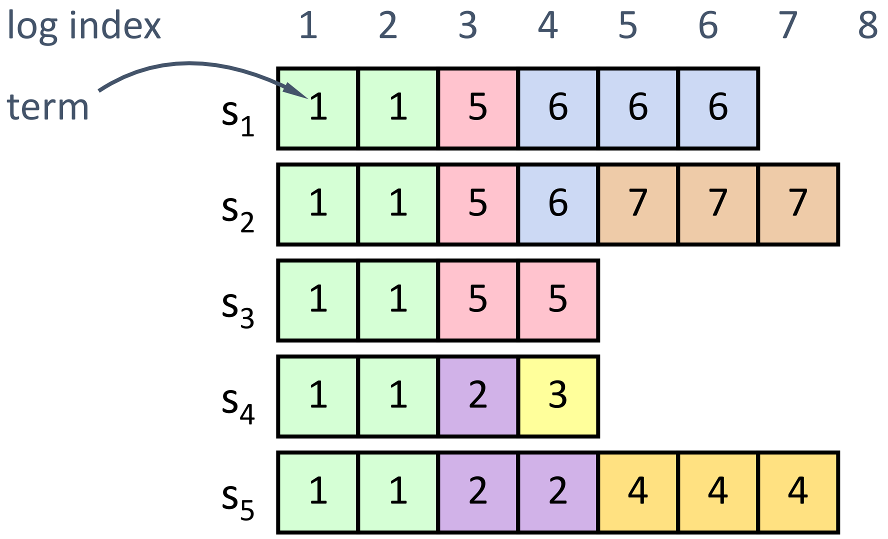
AppendEntries Consistency Check

- Each AppendEntries RPC contains index and term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency

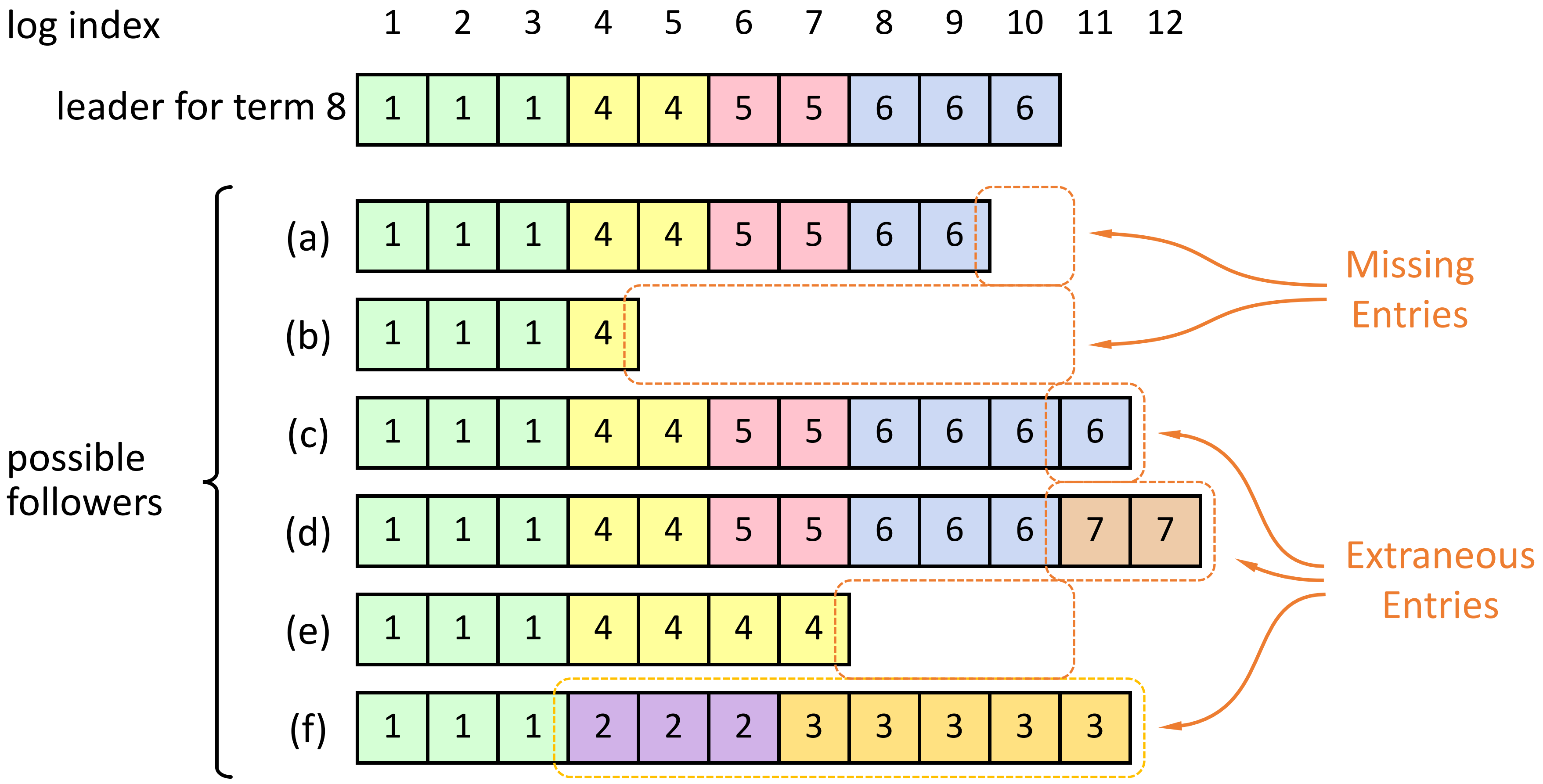


Leader Changes

- At beginning of new leader's term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - **Leader's log is "the truth"**
  - **Will eventually make follower's logs identical to leader's**
    - *Unless a new leader gets elected during the process.*
  - Multiple crashes can leave many extraneous log entries:

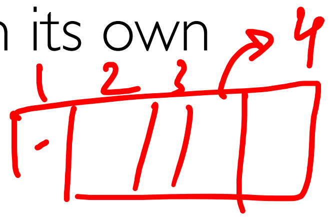


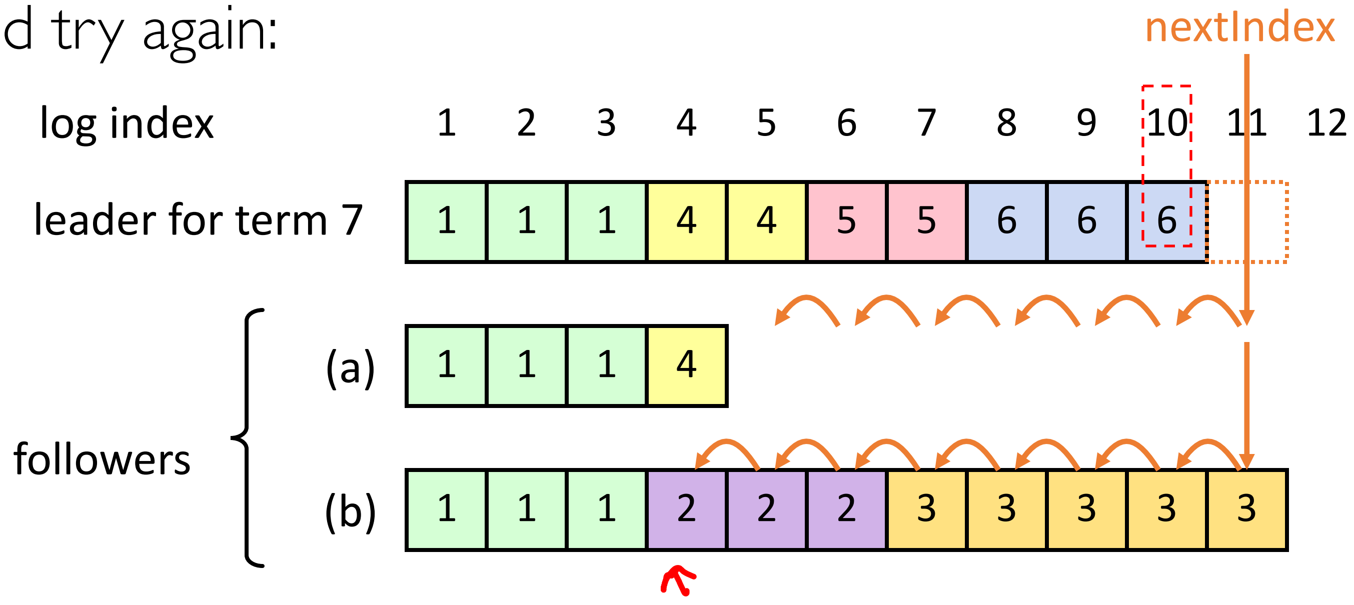
Log Inconsistencies



Repairing Follower Logs

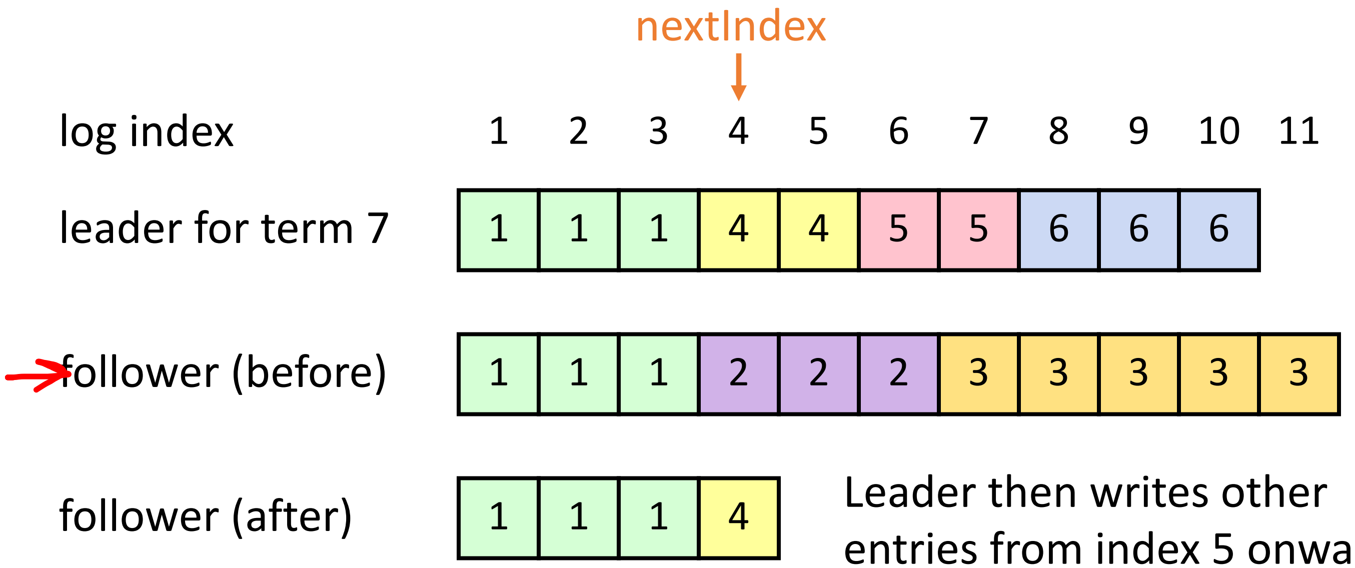
- New leader must make follower logs consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (leader's last index + 1)
- When AppendEntries consistency check fails, decrement nextIndex and try again:





Repairing Logs, cont'd

- When follower overwrites inconsistent entry, it deletes all subsequent entries:



Leader then writes other entries from index 5 onwards

Safety Requirement for log consensus

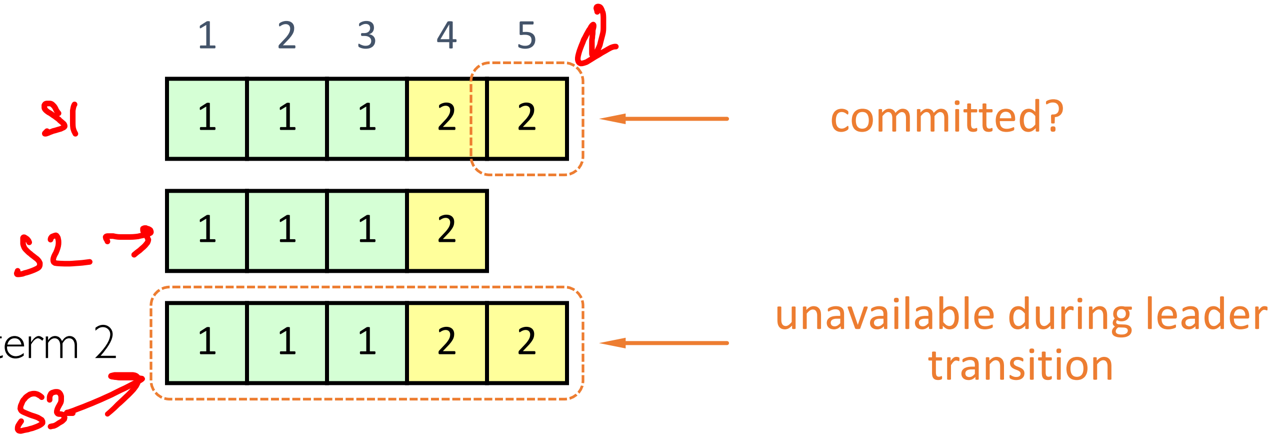
Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry.

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
- This guarantees the safety requirement
  - Leaders never overwrite entries in their logs
  - Only entries in the leader's log can be committed
  - Entries must be committed before applying to state machine

$$\begin{tikzcd}
& \text{Committed} \arrow[r] \arrow[dl, bend left] & \text{Present in future leaders' logs} \arrow[dr, bend right] & \\
\text{Restrictions on commitment} & & & \text{Restrictions on leader election}
\end{tikzcd}$$

Picking the Best Leader

- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more *up-to-date*":
    $(\text{lastTerm}_V > \text{lastTerm}_C) \;||$
    $(\text{lastTerm}_V == \text{lastTerm}_C) \;\&\&\; (\text{lastIndex}_V > \text{lastIndex}_C)$
  - Leader will have "most complete" log among electing majority



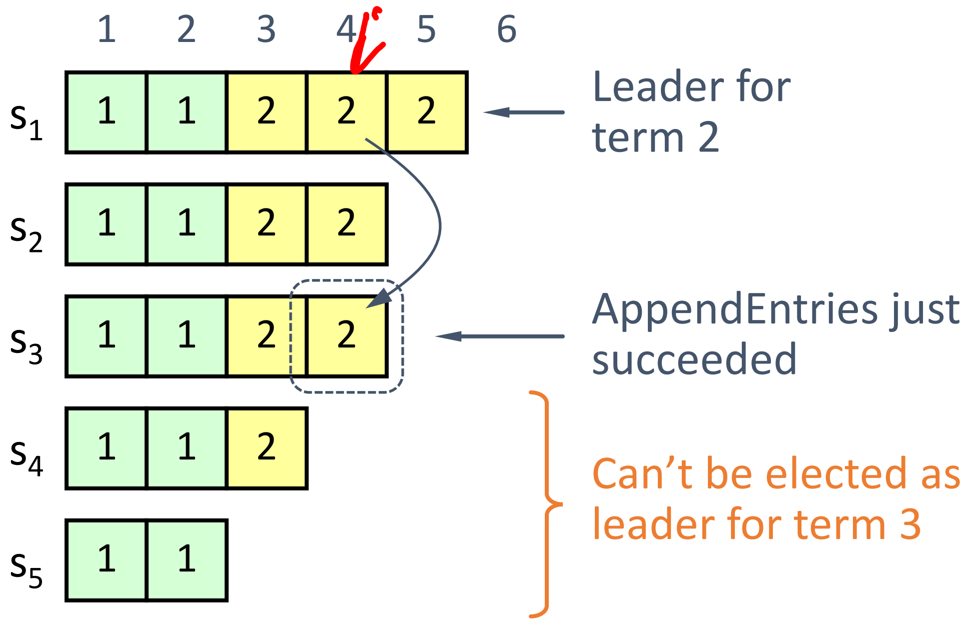
Election Basics: handling RequestVote RPCs

- Suppose a server S in term currentTerm has voted for process with id votedFor in that term.
- When it receives RequestVote RPC from process candidateId with term voteRequestTerm:

```
If voteRequestTerm < currentTerm
        Reply false, return.
If voteRequestTerm > currentTerm
        currentTerm = voteRequestTerm, votedFor = null
If (candidate's log is at least as up-to-date S's log) and (votedFor is null or candidateId)
        Grant vote, votedFor = candidateId
```

Committing Entry from Current Term

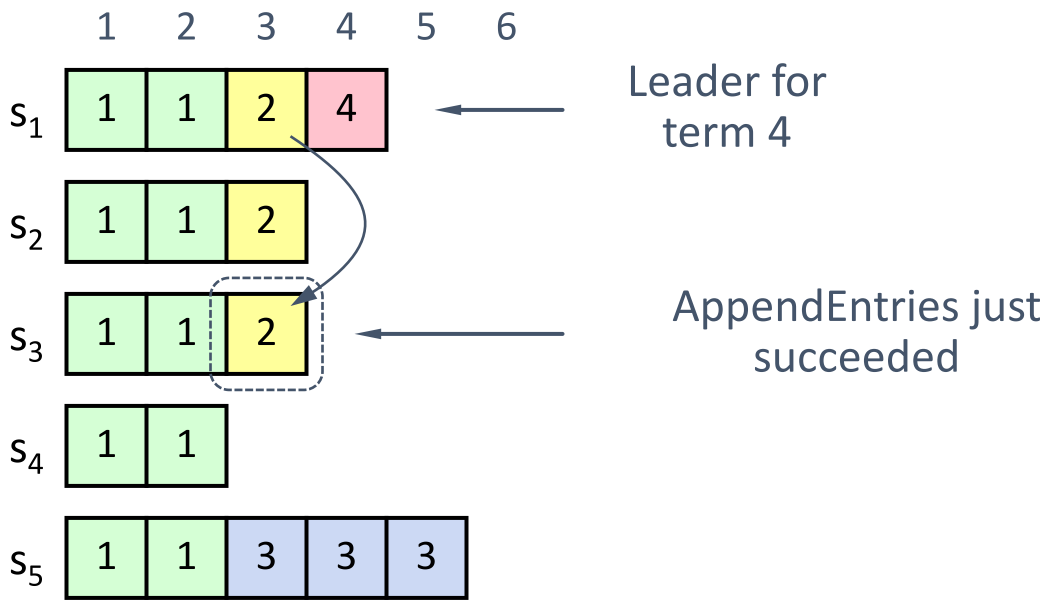
- *When can a leader commit entries?*



- Leader decides entry in term 4 is committed
  - Safe: leader for term 3 must contain entry 4
- What about committing entry in term 5?
- *Perhaps leader can commit an entry once replicated on majority of servers?*

Committing Entry from Earlier Term

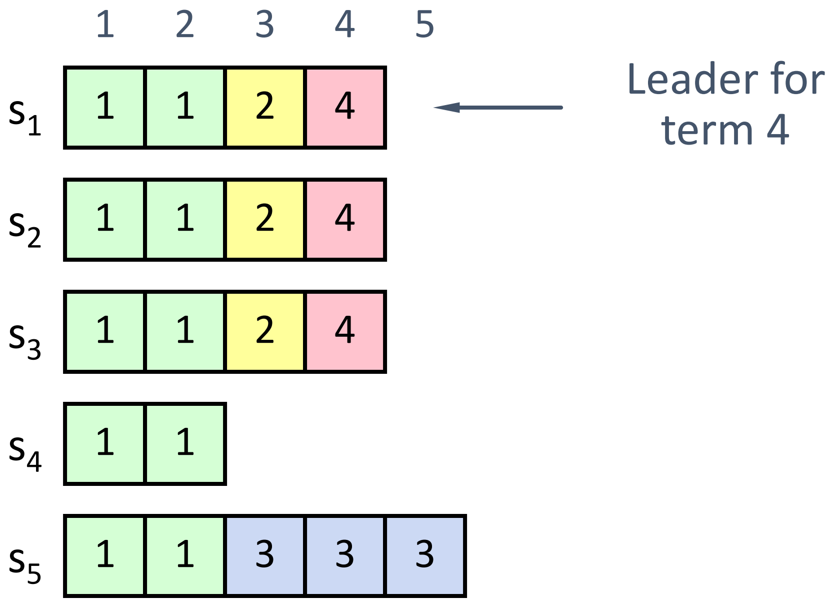
- Leader is trying to finish committing entry from an earlier term



- 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

- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - *At least one new entry from leader's current term must also be stored on majority of servers*
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Safety Requirement for log consensus

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

- Raft safety property:
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
- This guarantees the safety requirement
  - Leaders never overwrite entries in their logs
  - Only entries in the leader’s log can be committed
  - Entries must be committed before applying to state machine

$$\begin{tikzcd} & \textbf{Committed} \arrow[r] \arrow[dl, bend left] & \textbf{Present in future leaders' logs} \arrow[dr, bend right] & \\ \text{Restrictions on commitment} & & & \text{Restrictions on leader election} \end{tikzcd}$$

Raft Protocol Summary

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

More details in Raft paper

- Link on the course website.
- Play with the visualization at raft.github.io
- The concepts covered Section 6 and beyond are not in your syllabus.

Agenda for today

- Consensus
    - Consensus in synchronous systems
        - *Chapter 15.4*
    - Impossibility of consensus in asynchronous systems
        - *We will not cover the proof in details*
    - Good enough consensus algorithm for asynchronous systems:
        - *Paxos made simple, Leslie Lamport, 2001*
    - Other forms of consensus algorithm
        - Raft (log-based consensus)
        - Block-chains (distributed consensus)

Bitcoins

- Implement a *distributed* replicated state machine that maintains an *account ledger (= bank).*
  - No user should be able to "double-spend".
  - Need to know of all transactions to validate this.
  - Who does this validation? Cannot trust a single central authority.
    - Any participant (replica) should be able to validate.
  - All replicas must agree on the single history on transaction ordering.
- Scale to thousands of replicas distributed across the world.
- Allow old replicas to fail, new replicas to join seamlessly.
- Withstand various types of attacks.

Uses Blockchains for Consensus

- Why not use Paxos / Raft?
    - Need to scale to thousands of replicas across the world.
    - May not even know of all replicas a priori.
    - Participants may leave / join dynamically.
    - Paxos/Raft are ill-suited for such a setup.
        - Leader election in Raft or proposals in Paxos require communication with at least a majority of servers.
        - Require knowing the number of replicas.
        - ….

- *So how does blockchain work?*
    - Focus of today’s class. Only a high-level discussion.

Basic Idea

Transactions grouped into a *block* that gets added to the *chain* (history of transactions) by the "leader of that block".

How is this done? Next class.

MP2: Raft Leader Election and Log Consensus

- Lead TA: Jiangran Wang

- Objective:
  - Implement a leader-based consensus protocol for replicated state machine, that maintains log consensus even when nodes crash or get temporarily disconnected.

- Task:
  - Beef up a skeleton code provided to you to implement Raft leader election and log consensus.
  - We provide an emulation framework and a test suite.
  - Strive to pass all the test cases provided in our test suite.

MP2: Logistics

- Due on April 5th.
  - Late policy: Can use part of your 168hours of grace period accounted per student over the entire semester.
- Must be implemented in Go.
  - The framework we provide is in Go.
- Read the specification and the comments in the provided code carefully.
- **Start early!!**
  - MP2 is harder than MP1.