# Distributed Systems

## CS425/ECE428

*Instructor: Radhika Mittal*

# Logistics

- MP1 is due today.
- HW2 is due on Wednesday.
- MP2 has been released.
- HW3 will be released on Wednesday.
- Midterm 1 scores will be released in 1-2 weeks.

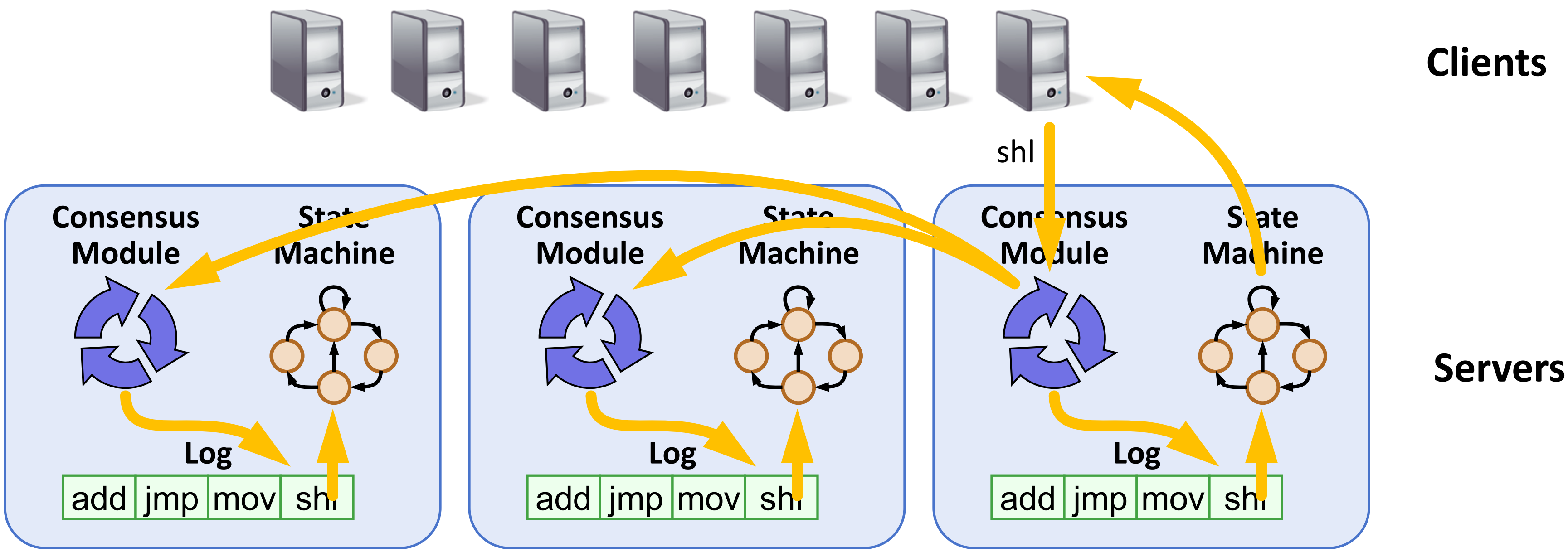
# Agenda for today

- 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: A Consensus Algorithm for Replicated Logs

Slides from 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
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

# Raft Overview

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

# Raft Overview

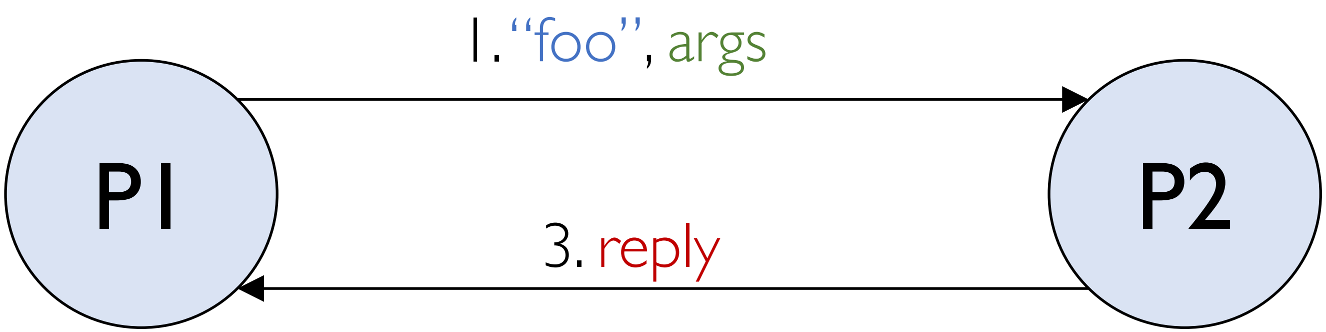
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

# Server States

- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive: issues no RPCs (requests), responds to incoming RPCs
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

# Quick Detour: RPCs

- Raft servers communicate via RPCs.
- What are RPCs?
  - Remote Procedure Calls: *procedure call between functions on different processes*
  - *Convenient programming abstraction.*

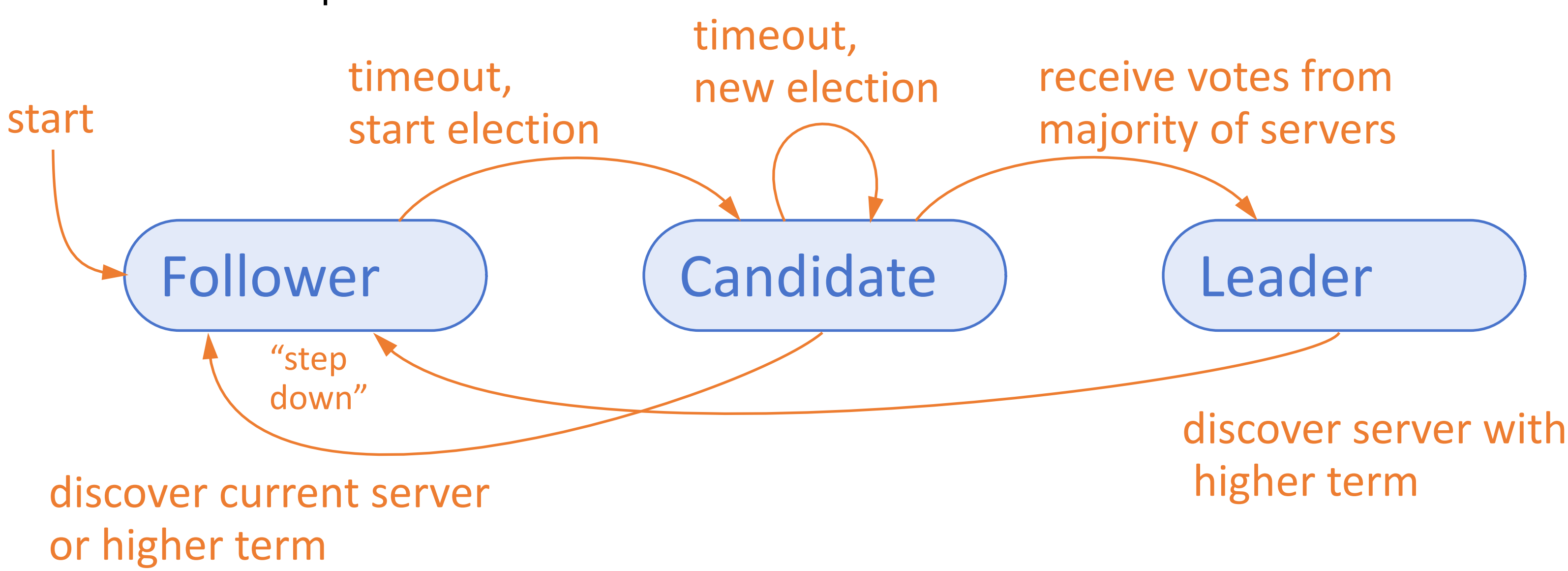



P2.call("foo", args, reply)

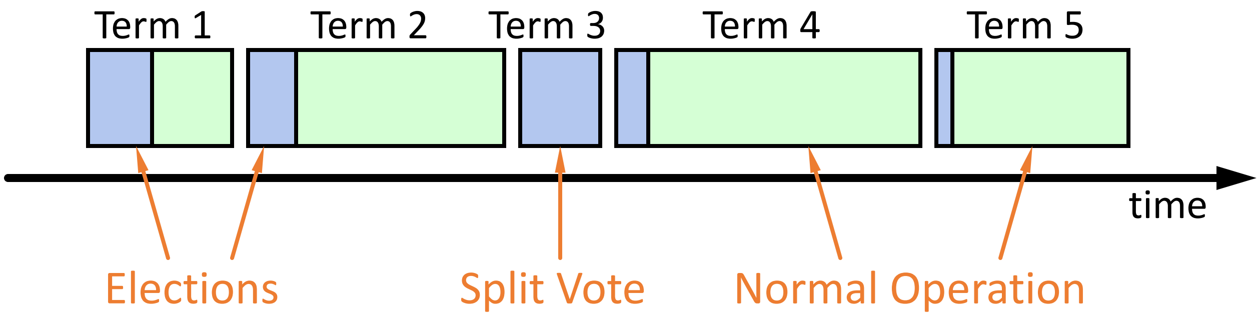
```
2. foo(args) {
    ....
    ....
    return reply
}
```

# Server States

- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive: issues no RPCs, responds to incoming RPCs
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers



# Terms




- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most 1 leader per term
- Some terms have no leader (failed election)
- Each server maintains current term value
- Key role of terms: identify obsolete information

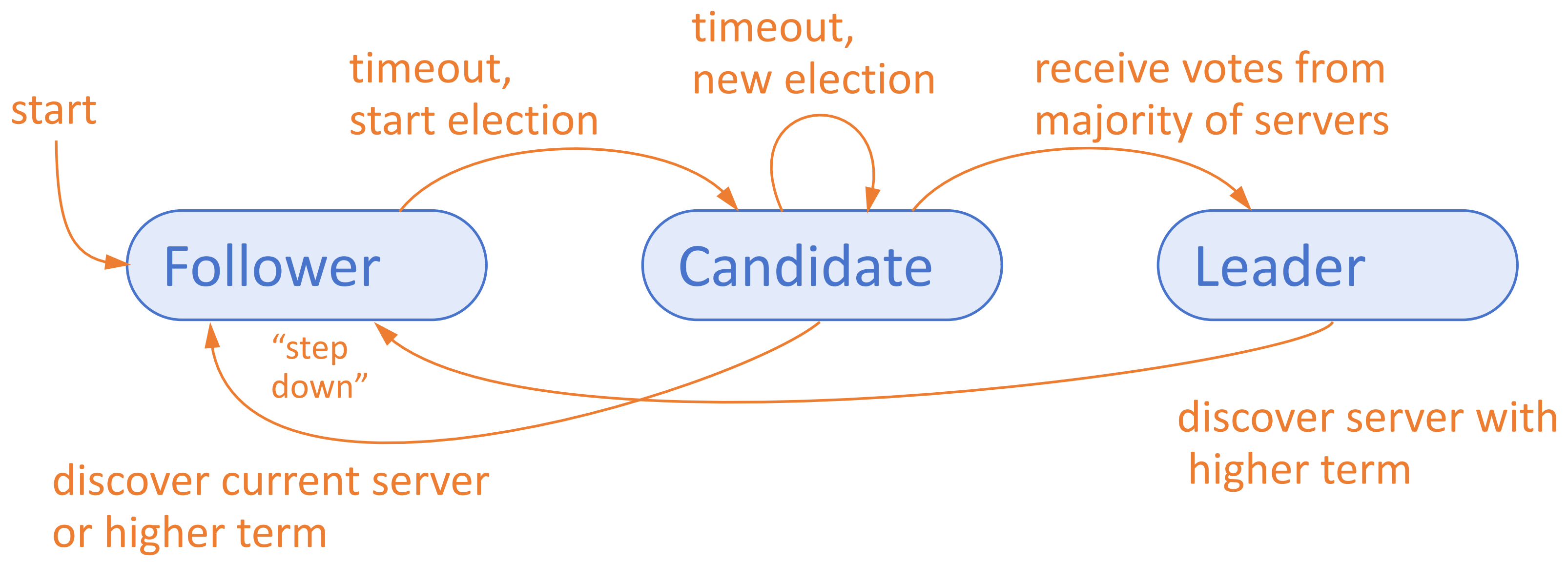
# Heartbeats and Timeouts

- Servers start up as followers.
- Followers expect to receive RPCs from leaders or candidates.
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority.
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower promotes itself to candidate and starts new election
  - Timeouts typically in range100-500ms
    - *Randomly* chosen in some range to reduce probability of split election.

# Election Basics

- On timeout:
  - Increment current term
  - Change to Candidate state
  - Vote for self
  - Send RequestVote RPCs to all other servers:
    1. Receive votes from majority of servers:
       - Become leader
       - Send AppendEntries heartbeats (RPCs) periodically to all other servers
    2. Receive RPC from valid leader (with same or higher term):
       - Return to follower state
    3. No-one wins election (election timeout elapses):
       - Increment term, start new election

# State Diagram Revisit



# Election Basics: handling RequestVote RPCs

- Suppose a server 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 (votedFor is null or candidateId)*

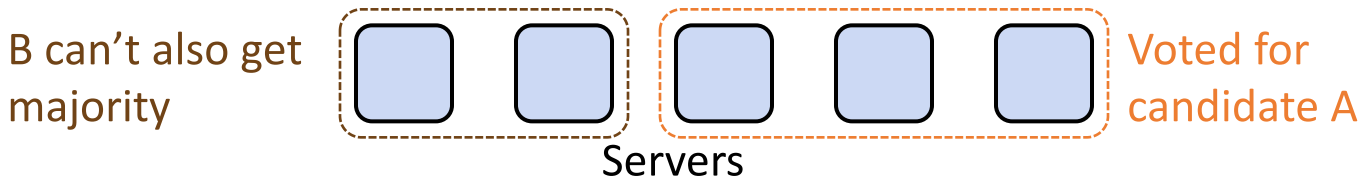
      //should not have voted for anyone else in that term

      Grant vote, votedFor = candidateId

      **_*we will extend on this condition later._**

# Elections, cont'd

- Safety: allow at most one winner per term




  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities *in same term*

- Liveness: some candidate must eventually win
  - Choose election timeouts randomly in [T, kT]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time
- *Safety is guaranteed. Liveness is not guaranteed.*

# Implication of terms

- Each term has at most one leader (safety condition).
- Terms always increase with time.
- If the latest term has an elected leader, majority of processes must have updated themselves to the latest term.
- Only the leader of the latest term can commit log entries (we will discuss this next).

# Raft Overview

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

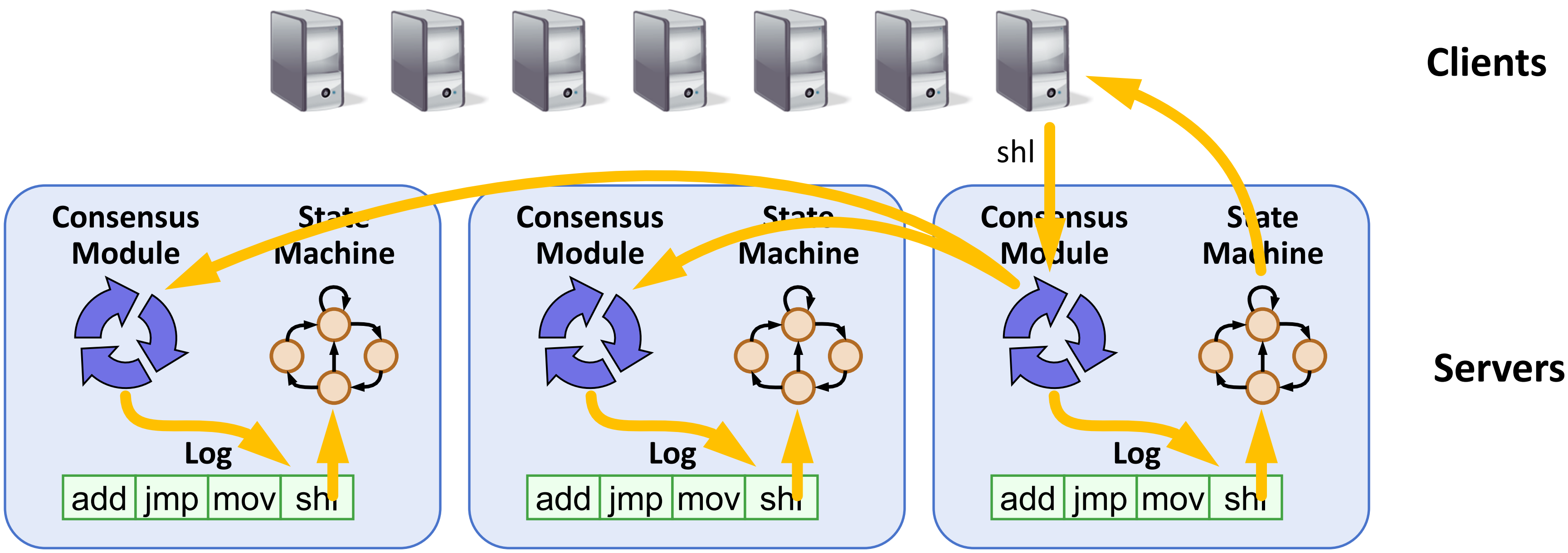
# Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

# Raft Overview

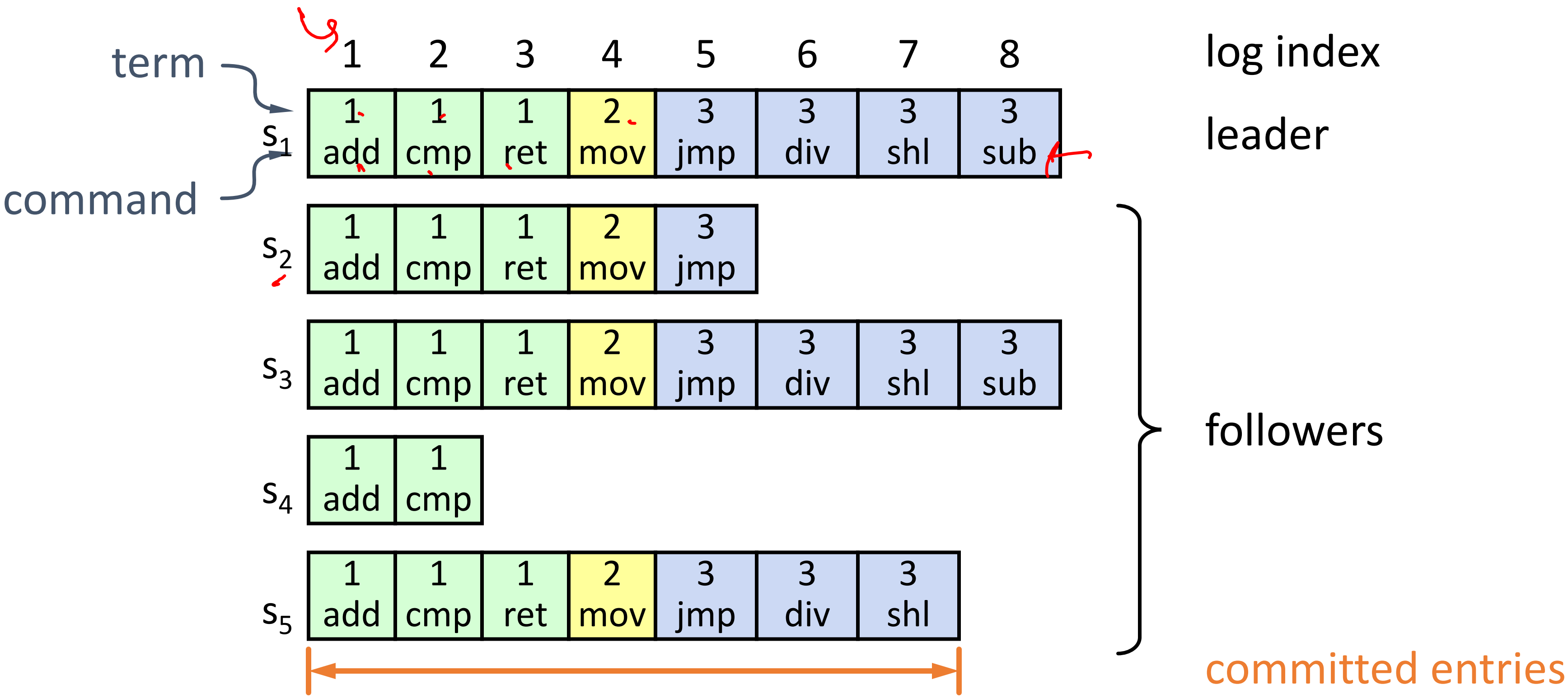
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

# Goal: Replicated Log




- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

# Log Structure




- 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.

# Normal Operation

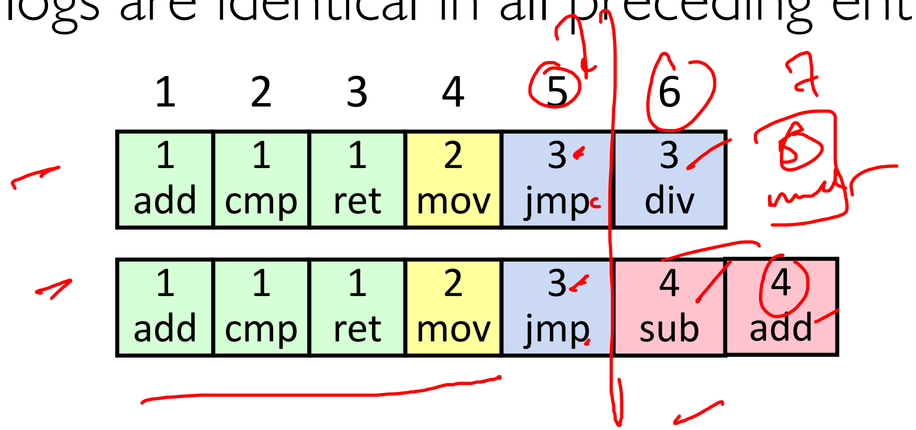
- Client sends command to leader
- Leader appends command to its log (*not yet committed*)
- Leader sends AppendEntries RPCs to followers
- Once new entry committed* (*we will discuss when and how*):
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

# Log Consistency

High level of coherency between logs:

Raft guarantees that:

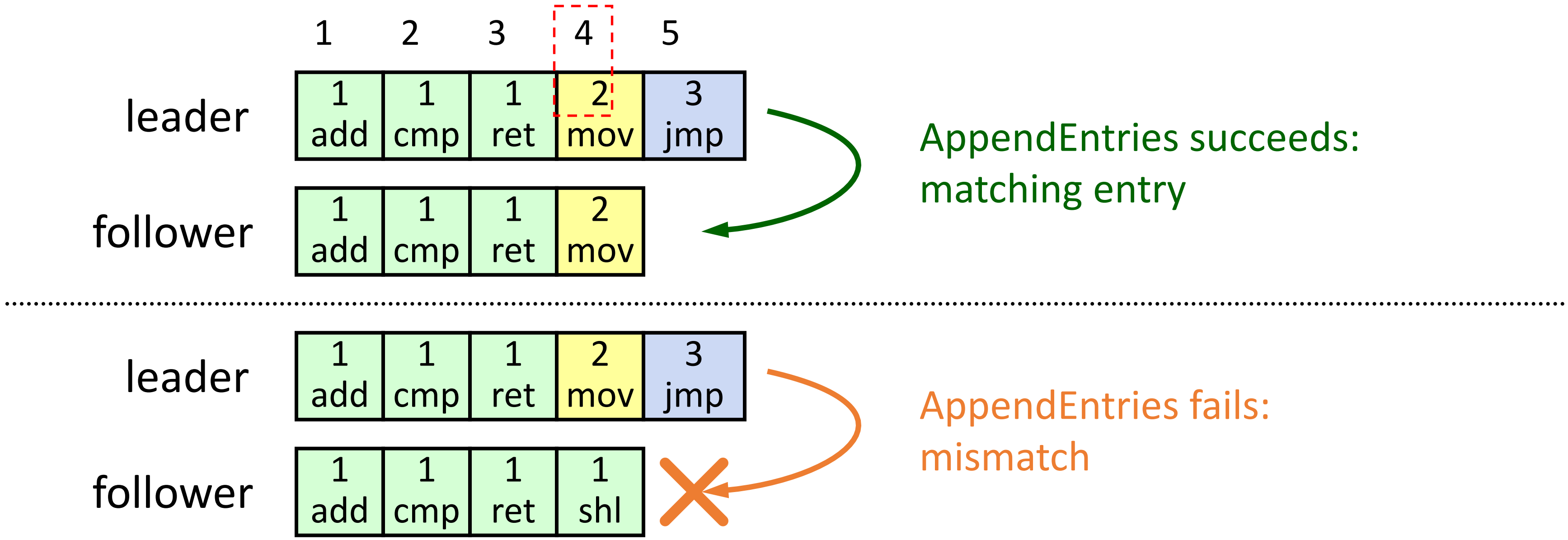
- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries




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

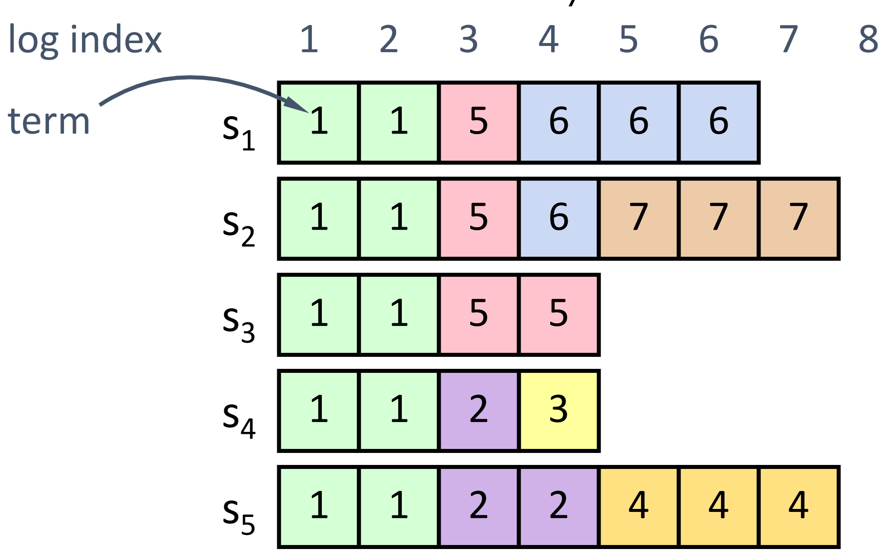
# AppendEntries Consistency Check

- 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

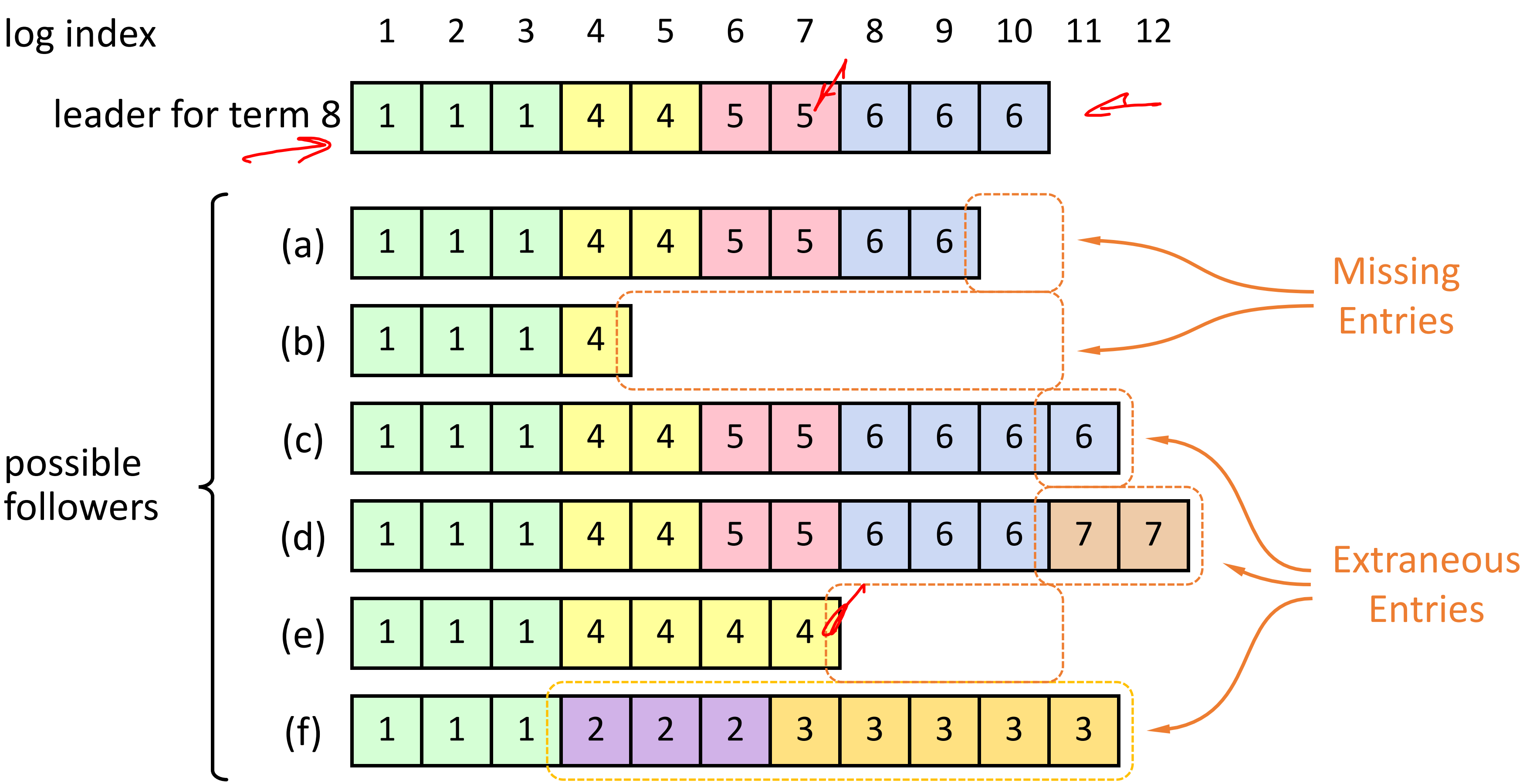


# Leader Changes

- 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:

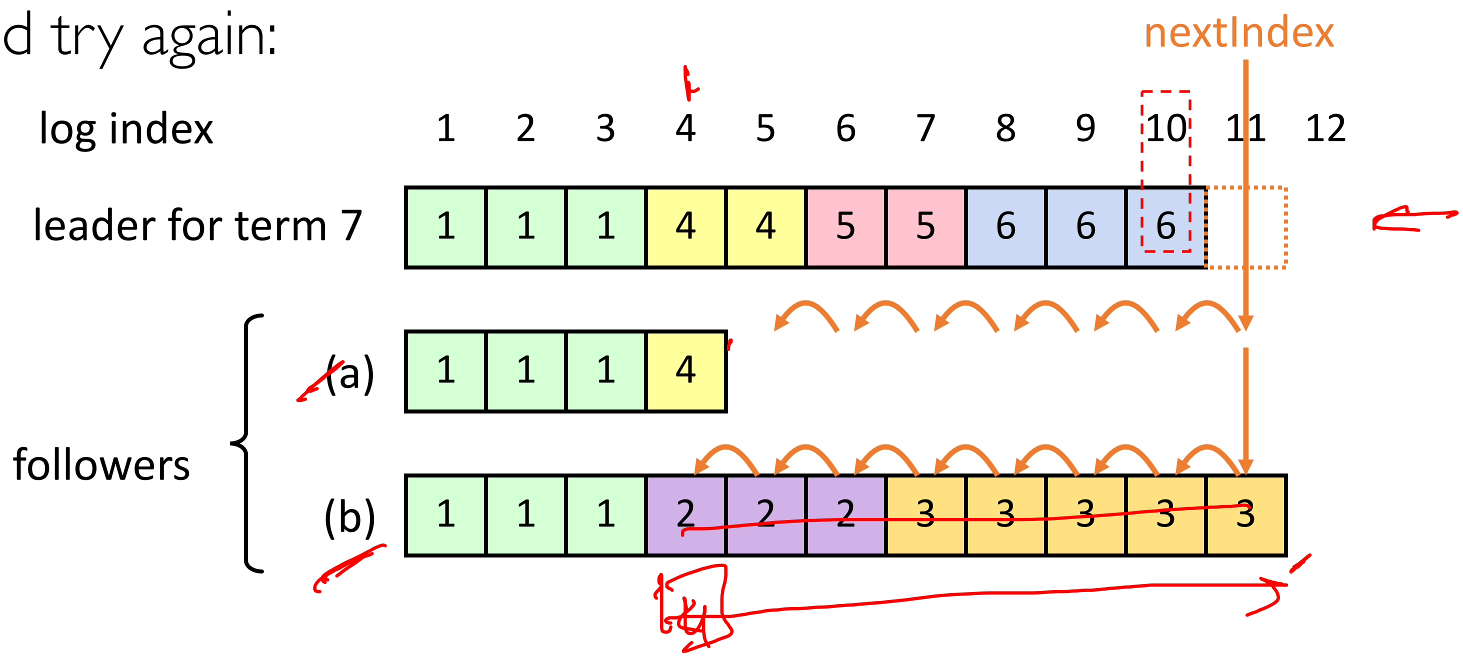


# Log Inconsistencies



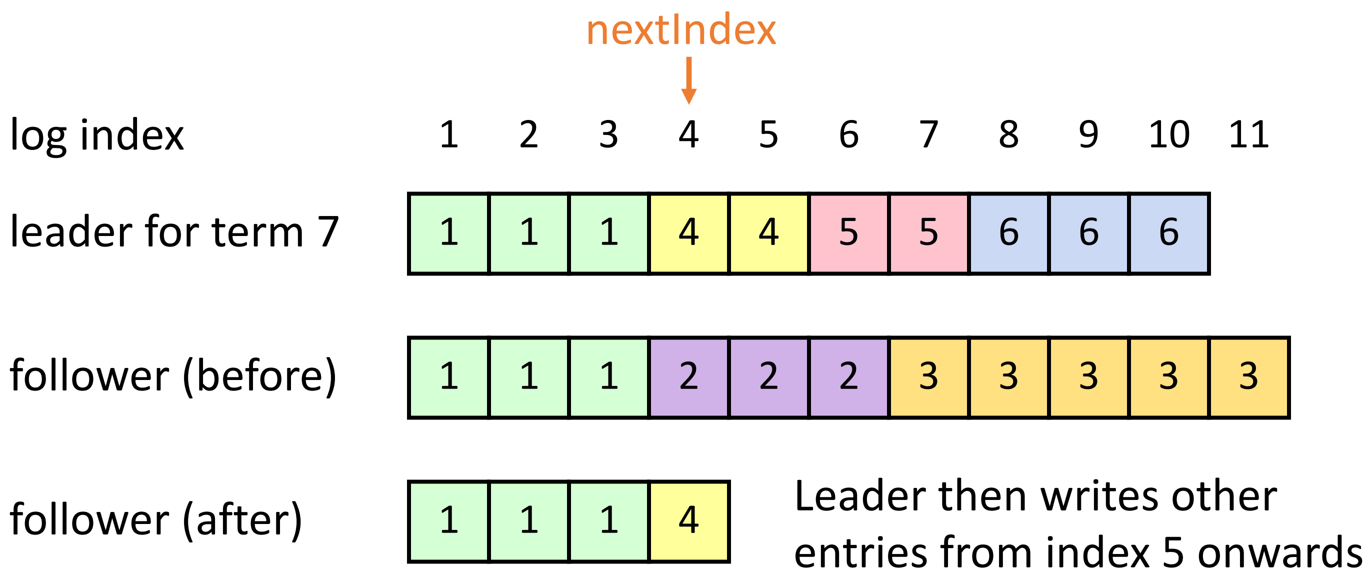
# Repairing Follower Logs

- 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 (1 + leader's last index)
- When AppendEntries consistency check fails, decrement nextIndex and try again:

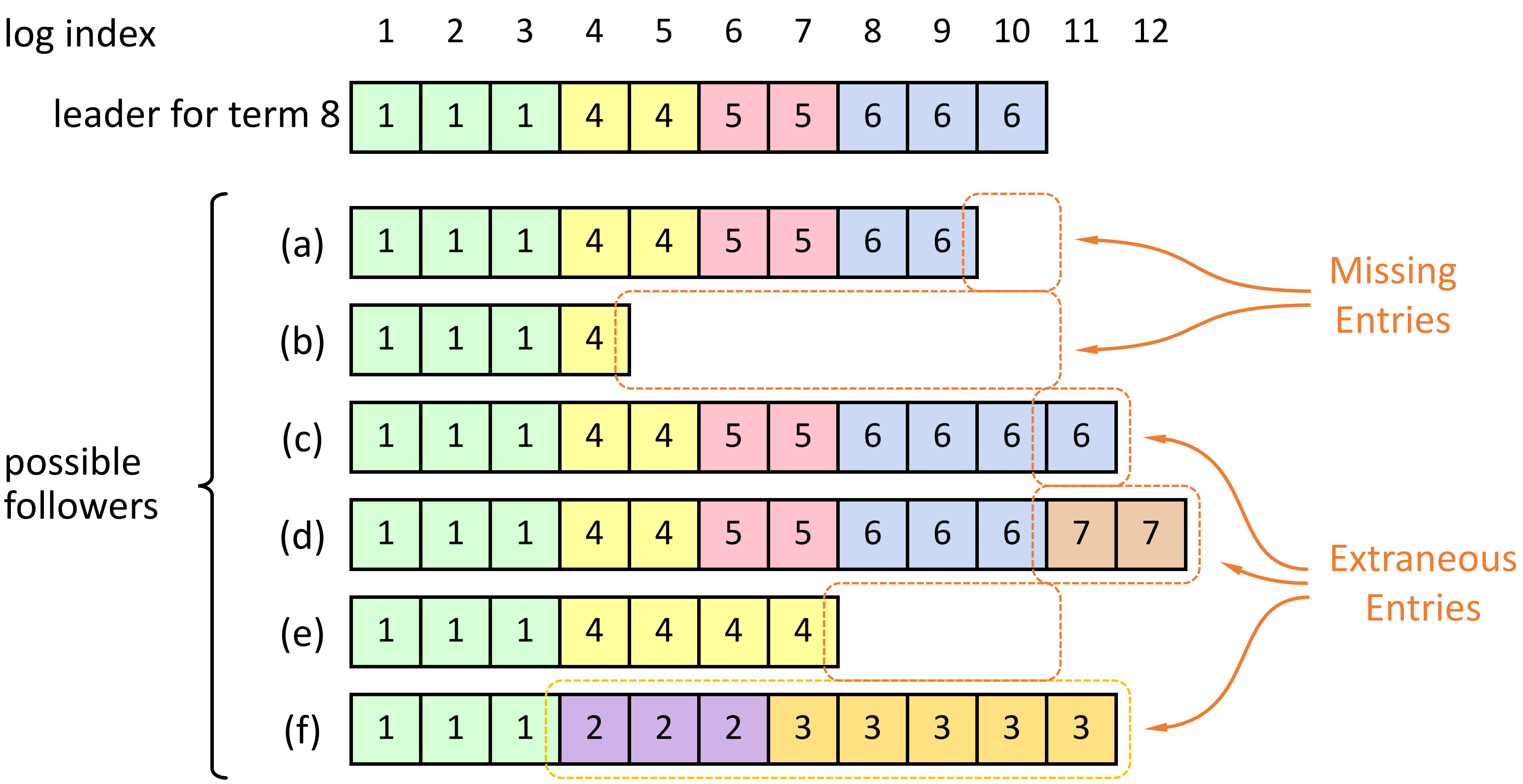


# Repairing Logs, cont'd

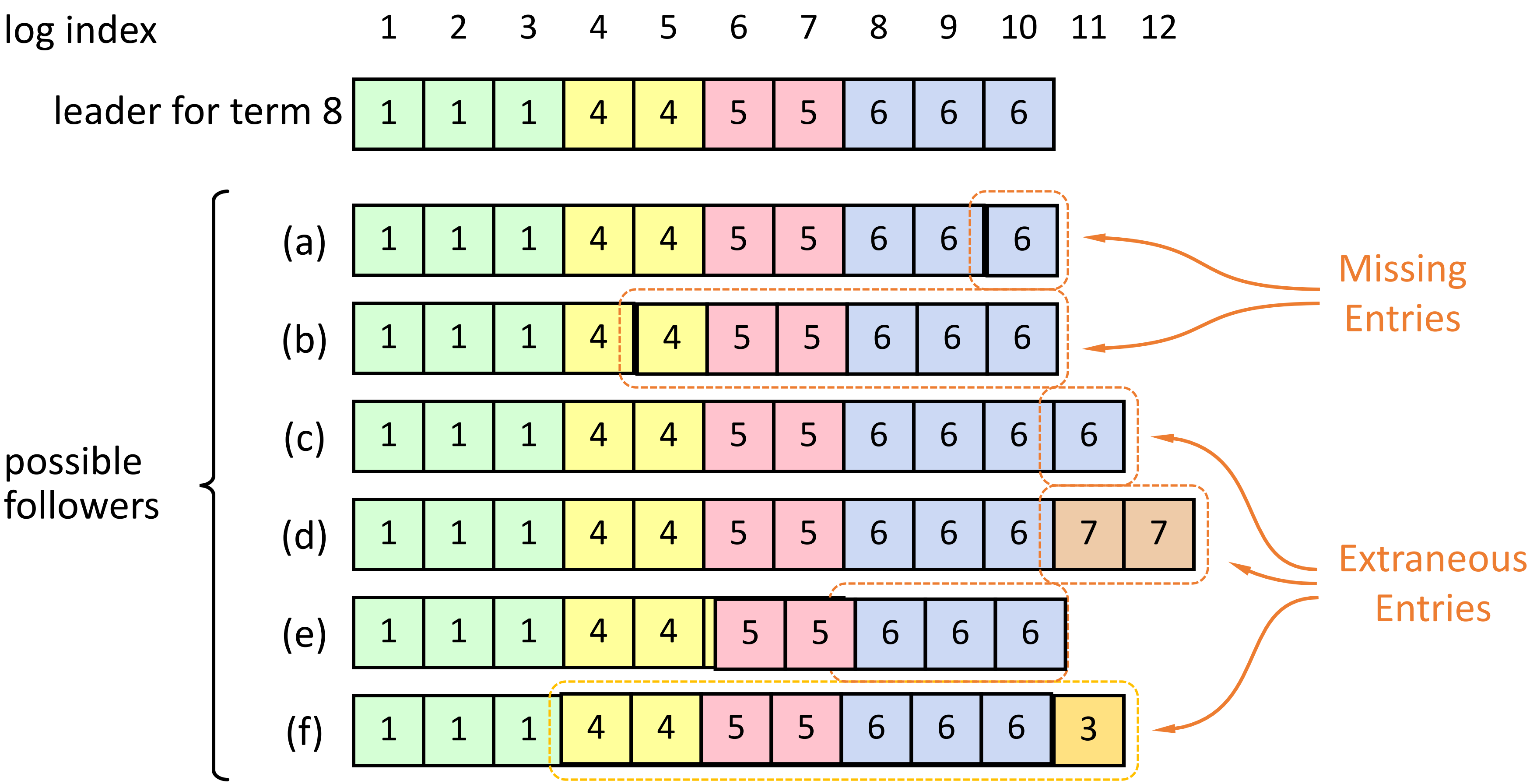
- When follower overwrites inconsistent entry, it deletes all subsequent entries:



# Log Inconsistencies



# Eventually make it consistent with leader



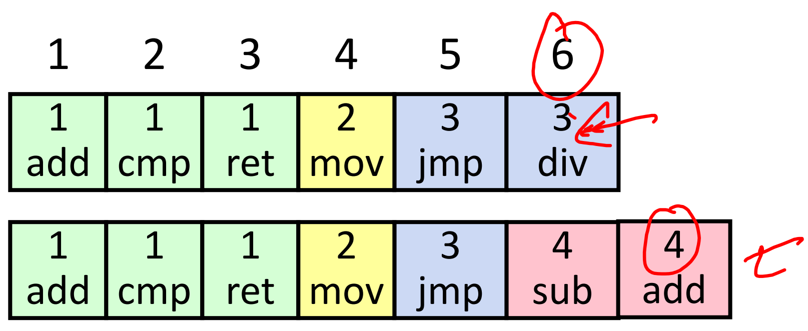

Leader changes in the process, can impact this.

# 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

# 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

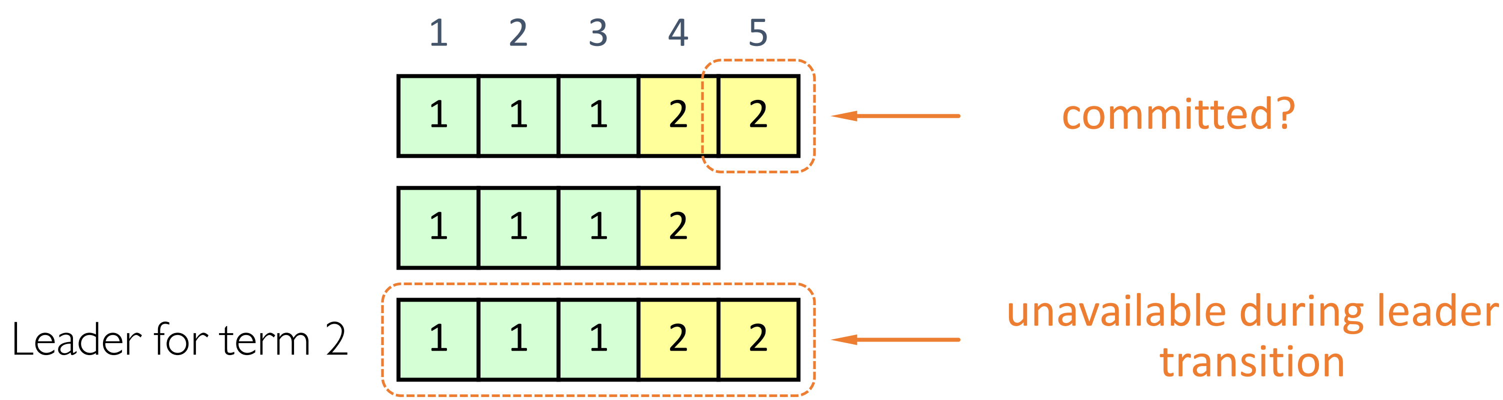
**Committed → Present in future leaders' logs**

Restrictions on commitment

Restrictions on leader election

# 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*":
    $(lastTerm_V > lastTerm_C) \,||$
    $(lastTerm_V == lastTerm_C) \,\&\&\, (lastIndex_V > lastIndex_C)$



# 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

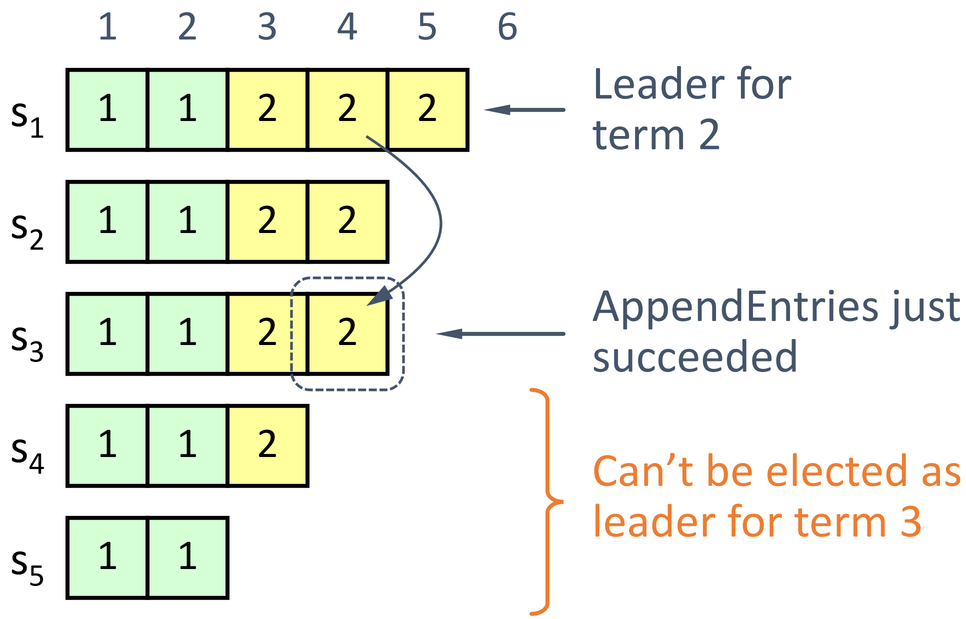
  currentTerm = voteRequestTerm, votedFor = null

  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

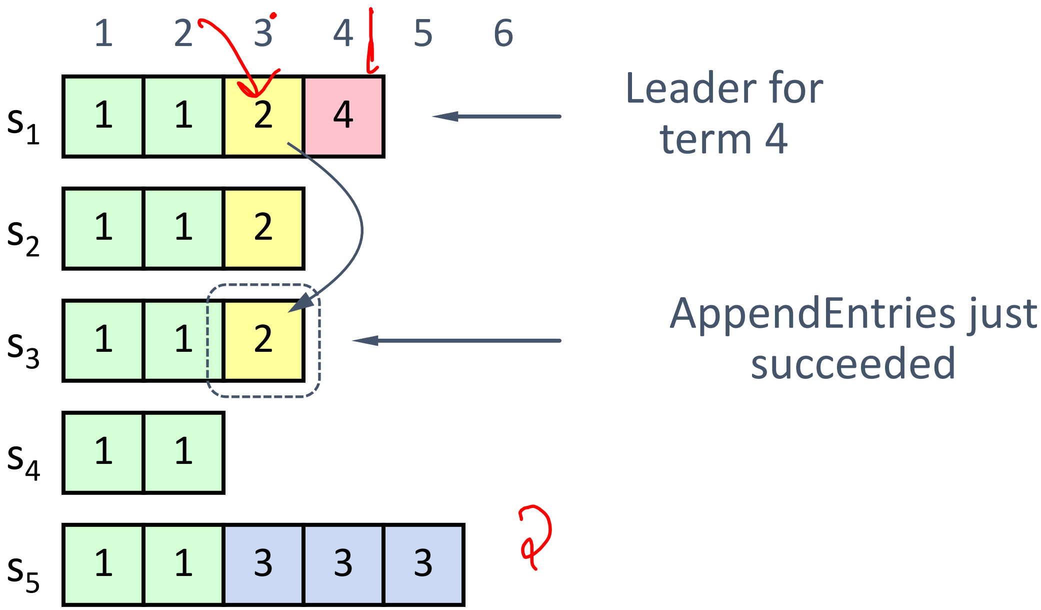
- *When can a leader commit entries?*




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

# Committing Entry from Earlier Term

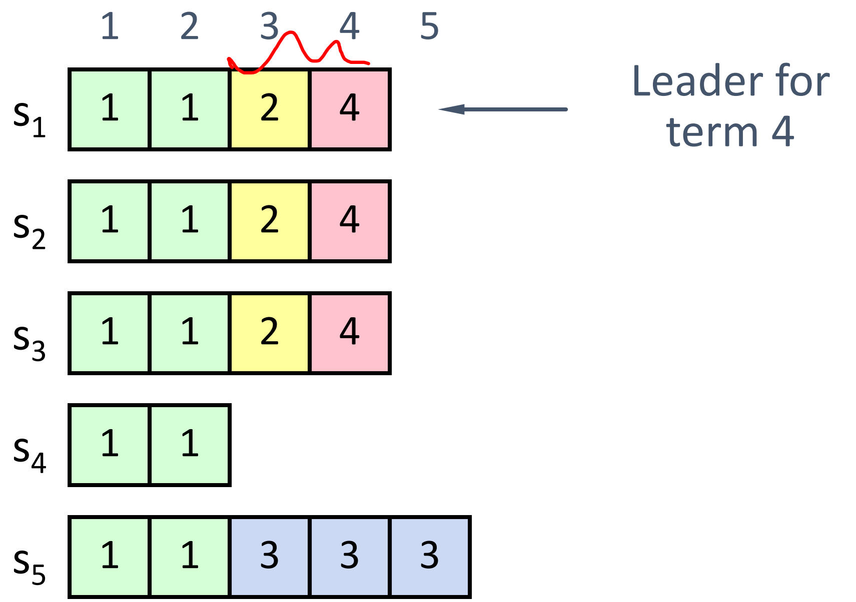
- Leader is trying to finish committing entry from an earlier term




- 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

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

Restrictions on commitment

Restrictions on leader election

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