CS 582: Distributed Systems

# Paxos and Raft

Dr. Zafar Ayyub Qazi

Fall 2024

Acknowledgments: Slides use some material from John Ousterhout (Stanford University

# Specific learning outcomes

By the end of today's lecture, you should be able to:

- ❑ Analyze how Paxos can be stuck in a livelock
- ❑ Analyze and evaluate Paxos for safety, liveness, and fault tolerance
- ❑ Explain how Raft works under normal operations
- ❑ Explain how log consistency is ensured under normal operations

# Recap: Paxos's Two-Phase Approach

- **Phase 1:** Broadcast Prepare Message
  - Find out about any chosen values
  - Block older proposals that have not yet been completed
- **Phase 2:** Broadcast Accept Message
  - Ask acceptors to accept a specific value

# Paxos

### Proposers

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) When responses received from majority:
   - Any rejections (result > n)?  goto (1)
   - Otherwise, **value is chosen**

### Acceptors

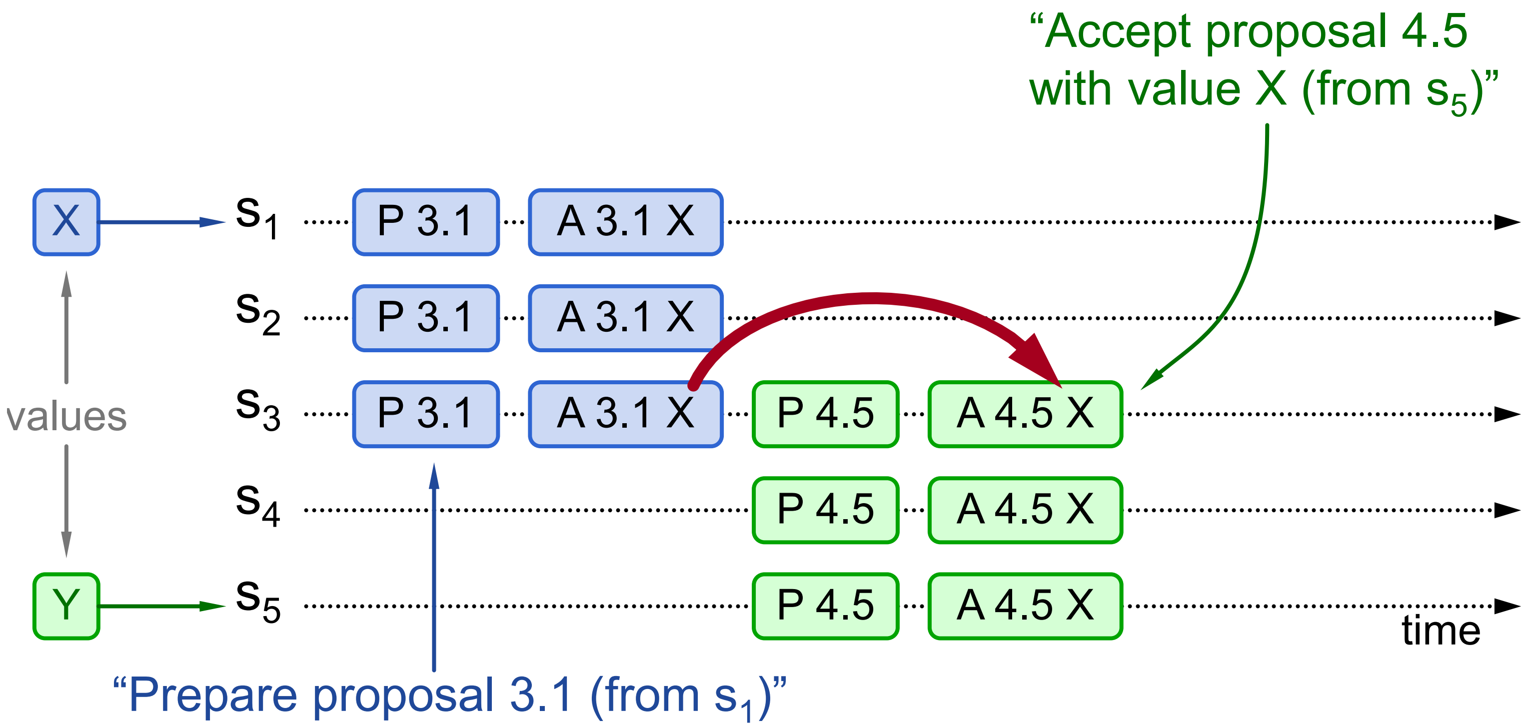
3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)
6) Respond to Accept(n, value):
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Return(minProposal)

**Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)**

# Paxos Examples

## 1. Previous value already chosen

- New proposer will find it and use it




### Paxos Protocol

**Proposers**

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

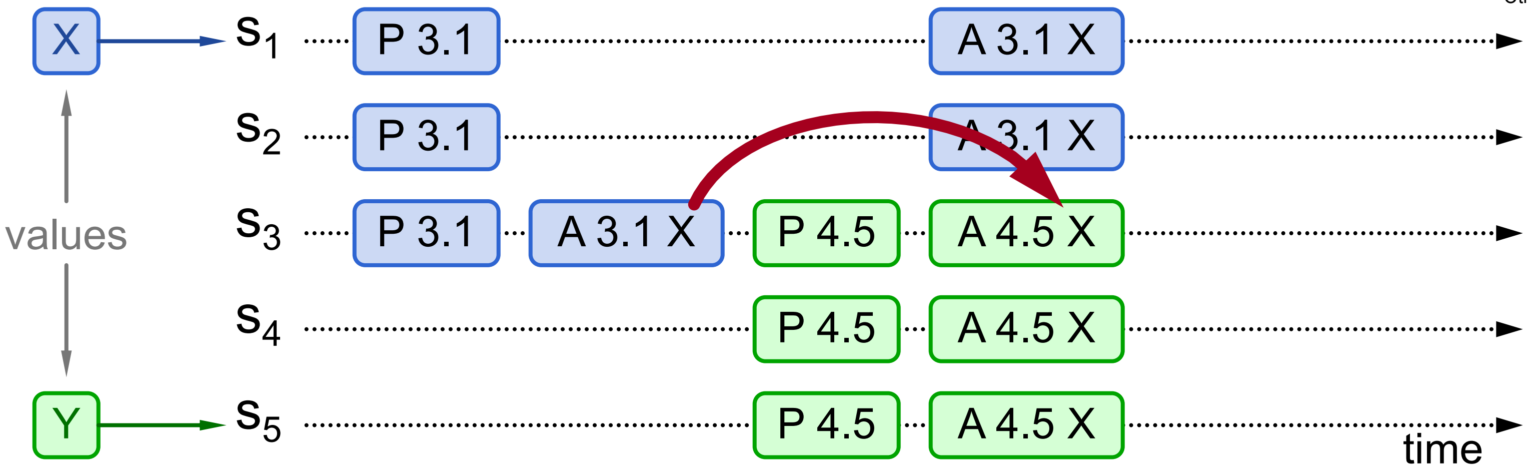
**Acceptors**

3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)
6) Respond to Accept(n, value):
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Return(minProposal)

# Paxos Examples

## 2. Previous value not chosen, but new proposer sees it

- New proposer will use existing value
- Both proposers can succeed




### Paxos Protocol

**Proposers**

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

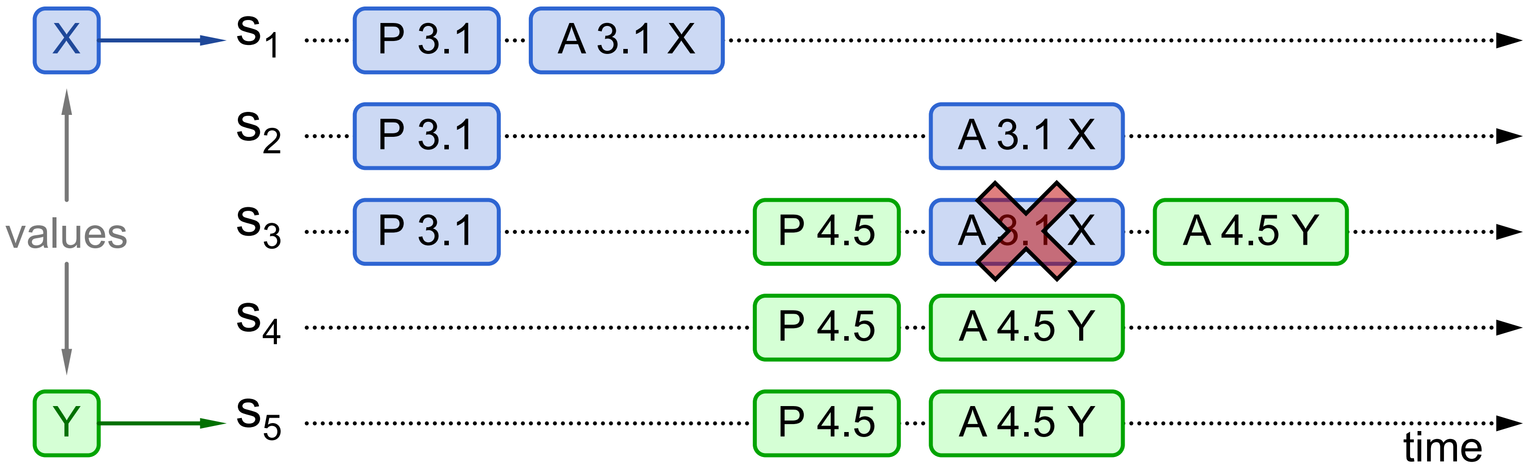
**Acceptors**

3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)
6) Respond to Accept(n, value):
   - If n ≥ minProposal then acceptedProposal = minProposal = n acceptedValue = value
   - Return(minProposal)

# Paxos Examples

## 3. Previous value not chosen, new proposer doesn't see it

- New proposer chooses its own value
- Older proposal blocked




### Paxos Protocol

**Proposers**

1) Choose new proposal number n
2) Broadcast Prepare(n) to all servers
4) When responses received from majority:
   - If any acceptedValues returned, replace value with acceptedValue for highest acceptedProposal
5) Broadcast Accept(n, value) to all servers
6) When responses received from majority:
   - Any rejections (result > n)? goto (1)
   - Otherwise, **value is chosen**

**Acceptors**

3) Respond to Prepare(n):
   - If n > minProposal then minProposal = n
   - Return(acceptedProposal, acceptedValue)
6) Respond to Accept(n, value):
   - If n ≥ minProposal then
     acceptedProposal = minProposal = n
     acceptedValue = value
   - Return(minProposal)

# Other points to note

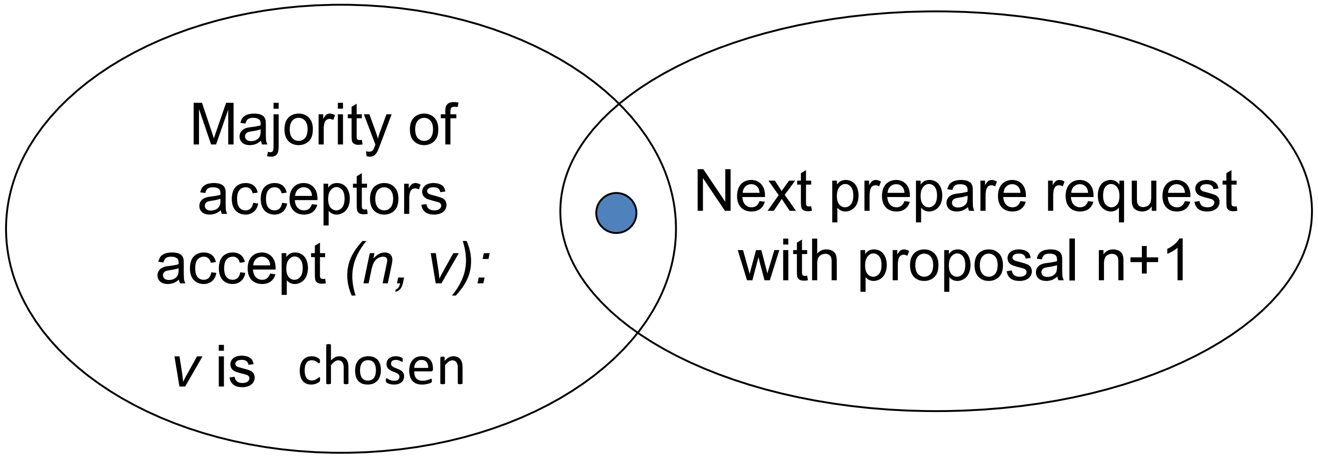
- Only proposer knows which value has been chosen
- If other servers want to know, must execute Paxos with their own proposal

# Paxos

- Safety?
- Liveness?
- Performance?

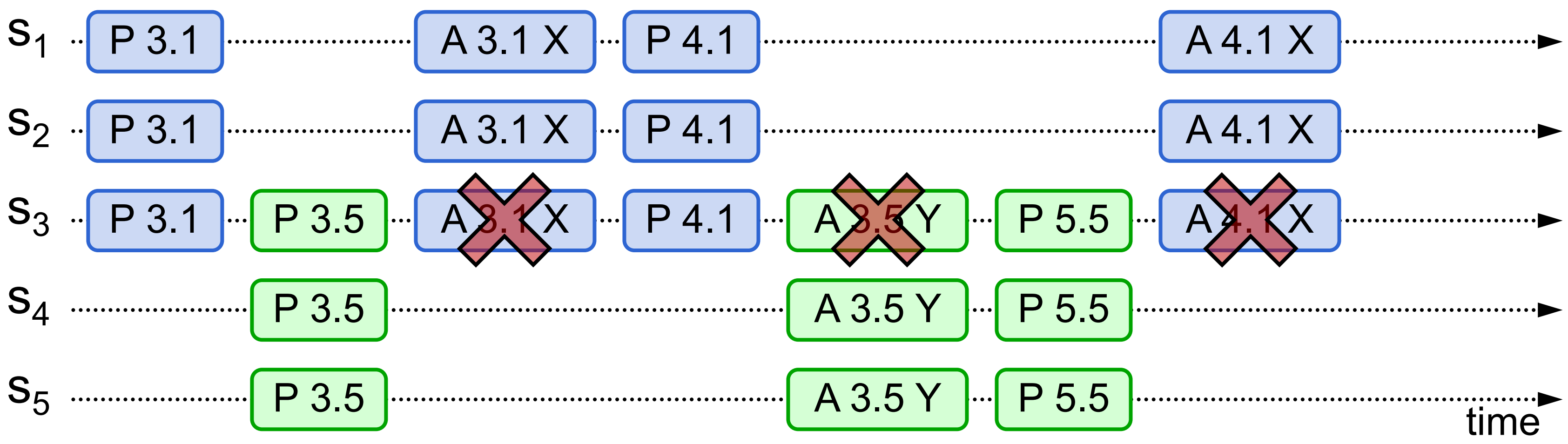
# Safety

- Intuition: if a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v



# Liveness

- Competing proposers can livelock:




- One solution: randomized delay before restarting
  - Give other proposers a chance to finish choosing
- Can use leader elections

# Performance?

- > 2 Round Trip Network Delays + Multiple Disk Writes

# Paxos Fault tolerance

- If there can be $f$ fail-stop failures in a system
- What are the minimum number of nodes Paxos needs to ensure consensus is reached?

- $2f + 1$
  - Each operation uses at least $f + 1$ nodes
  - Overlapping quorums

# Paxos: Summary

- Safety: Never violated
- On Liveness
  - If things go well sometime in the future (messages and failures, etc.), there is a good chance consensus will be reached.
- FLP result still applies:
  - Paxos is not guaranteed to reach a consensus (ever or within a bounded time)

# Paxos Problems

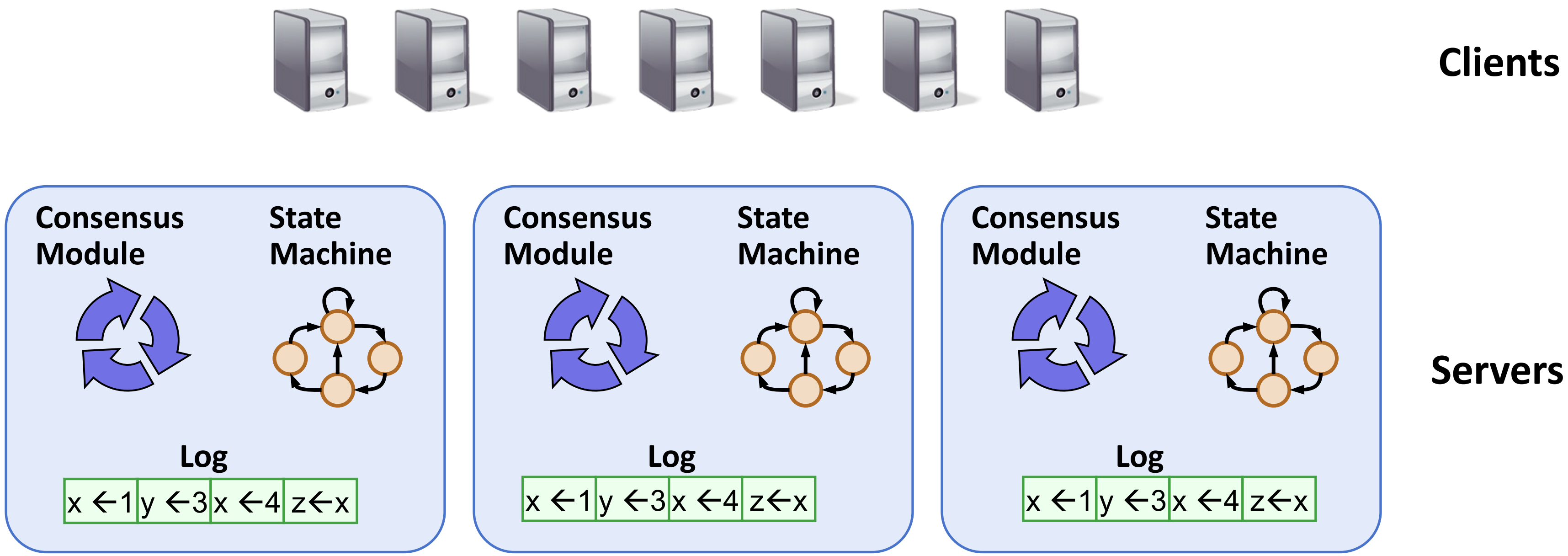
- Basic Paxos solves the problem for a single value
  - However, non-trivial to extend to Multi-Paxos. No agreement on the details of Multi-Paxos
  - We will discuss Multi-Paxos after we complete our discussion of Raft
- Doesn't fully address liveness
- Does not discuss cluster membership management

Led John Ousterhout and his PhD student Diego Ongara, to design a new consensus algorithm with **understandability as a primary design goal**

# Next …

- Raft: In Search of an Understandable Consensus Algorithm

# Raft Goal → Replicated Log




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

# Approaches to Consensus

Two general approaches to consensus:

- Symmetric, leader-less (like Replicated-Write Protocols):
  - All servers have equal roles
  - Clients can contact any server
- Asymmetric, leader-based (like Primary-backup):
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
- Raft uses a leader:
  - Decomposes the problem (normal operation, leader changes)
  - Simplifies normal operation (no conflicts)

# Raft Overview

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

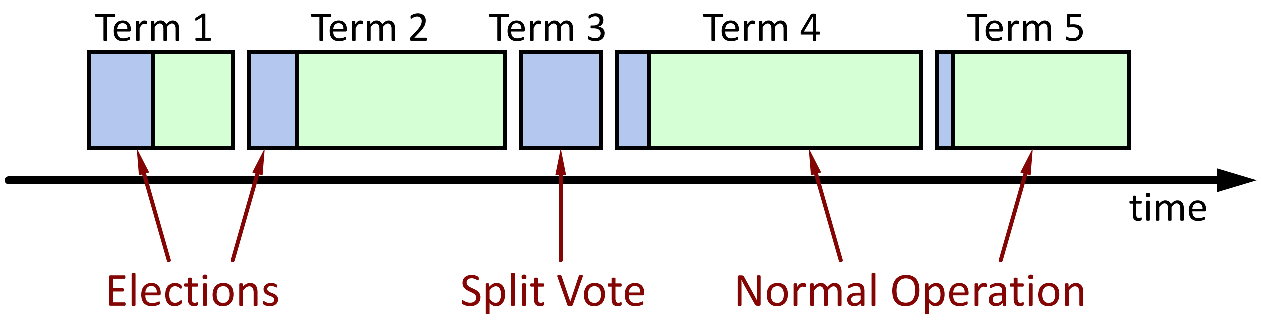
# Raft Overview

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

# 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: Only responds to incoming requests
- Candidate: Used to elect a new leader

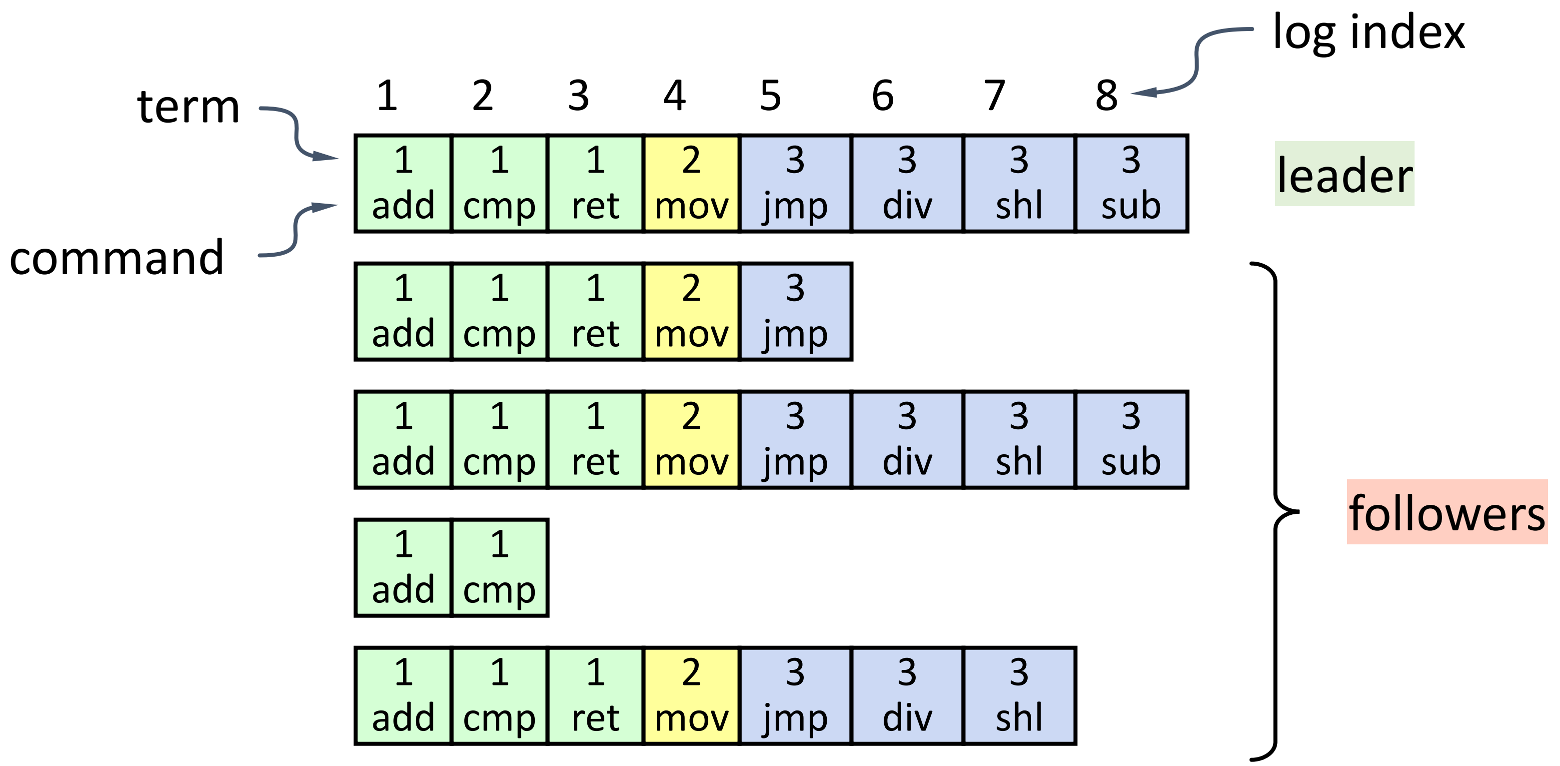
# Terms




- 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

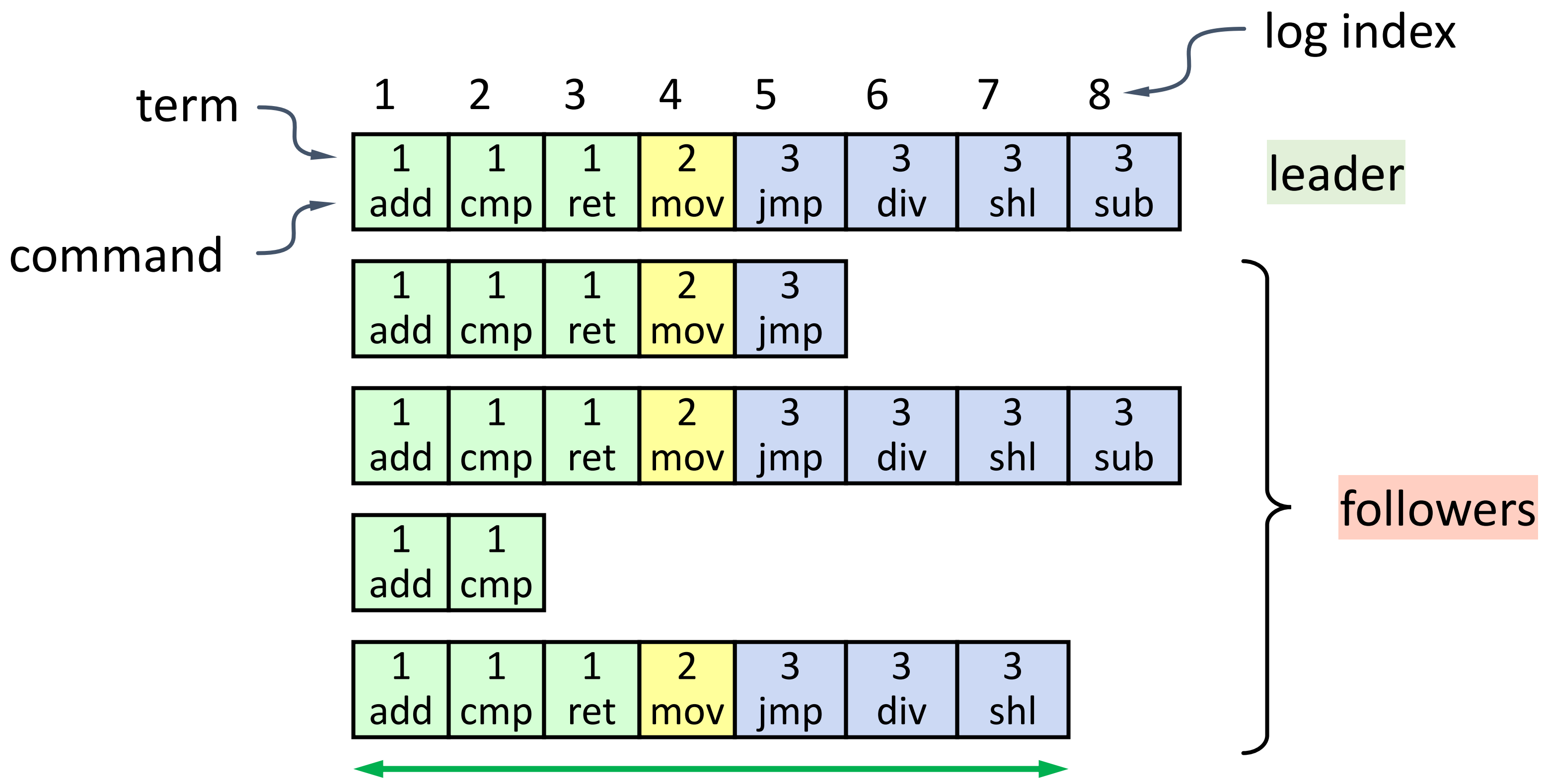
# Log Structure

# Log Structure




- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes

# 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
  - Will eventually be executed by state machines

# Normal Operations

- 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, 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 AppendEntries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

# Log Consistency in Raft?

- What can we say about the server logs under normal operation?
  - Would all server logs look identical?

- If a given entry is committed, what can we say about the preceding entries in the log?

# Log Consistency Property in Raft

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

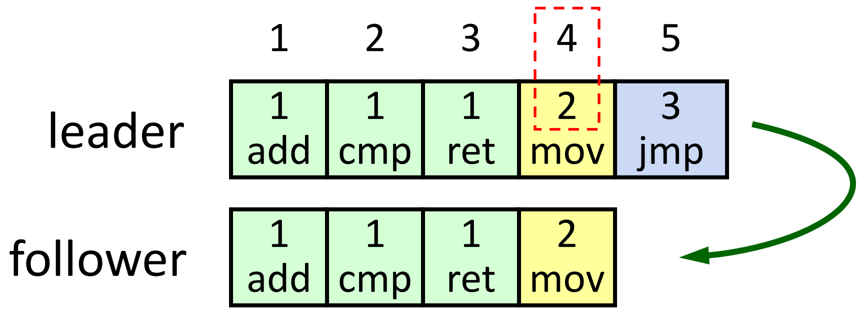
| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div |
| 1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 4 sub |

- If an entry is committed, all preceding entries are also committed

# How is log consistency ensured?

# AppendEntries Consistency Check

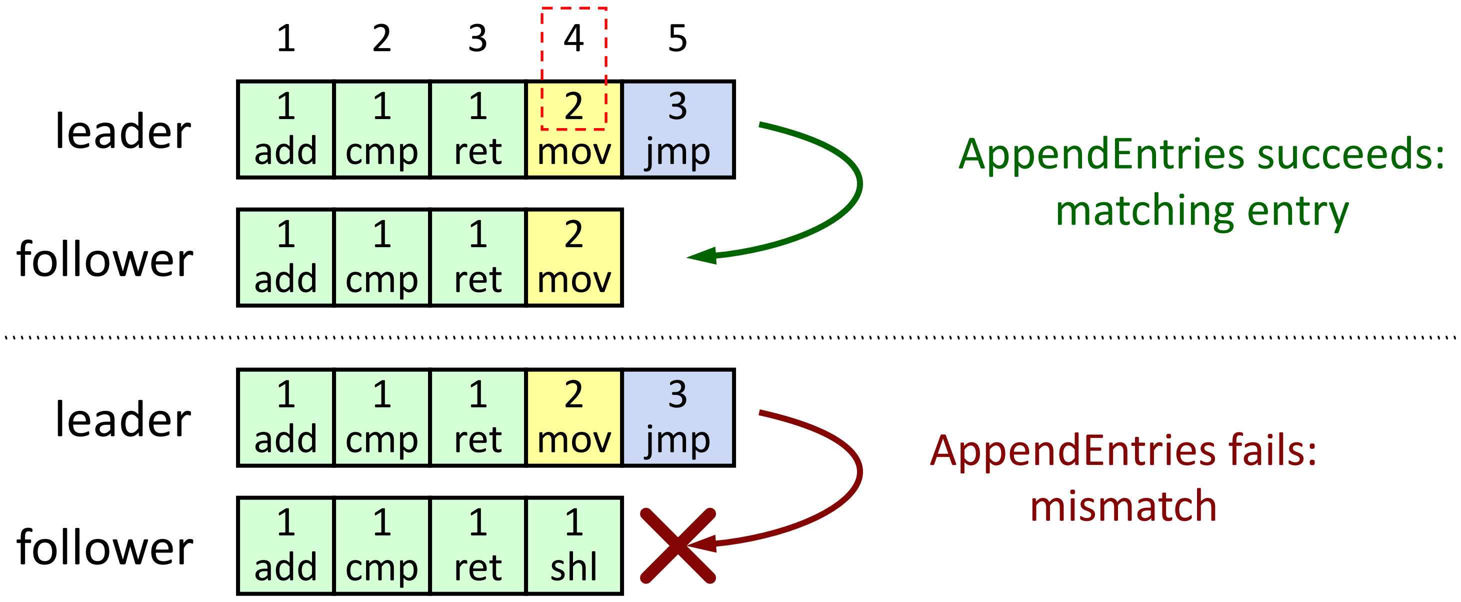
- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request




AppendEntries succeeds:
matching entry

# AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request



# Summary: Normal Operations

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry is committed:
  - 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

# Discussion so far …

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