CS 582: Distributed Systems

Raft: In Search of an Understandable Consensus Algorithm

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:

- ❑ Explain how Raft works under normal operations
- ❑ Explain and analyze how log consistency is ensured under normal operations
- ❑ Analyze what could go wrong as the leader changes
- ❑ Explain the safety requirements in Raft
- ❑ Explain and analyze how Raft provides safety and consistency after a leader changes
- ❑ Explain how Raft prevents an old leader from causing inconsistency in the system
- ❑ Explain how clients interact with the Raft system

Challenges with Paxos

- Basic Paxos solves the problem for a single-value
- However, extending to Multi-Paxos is hard
    - No agreed-upon algorithm for Multi-Paxos

Google Chubby Implementers:

There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. . . . the final system will be based on an un- proven protocol

Why Raft?

- Easier to understand and reason about
  - Relative to Paxos
- Fully functional and complete open-source implementations
- Used by several companies
  - Uber, Alibaba Cloud, IBM, Redis Labs, Baidu, Cisco, CoreOS, Cockroach Labs, PingCAP, VMWare, etc

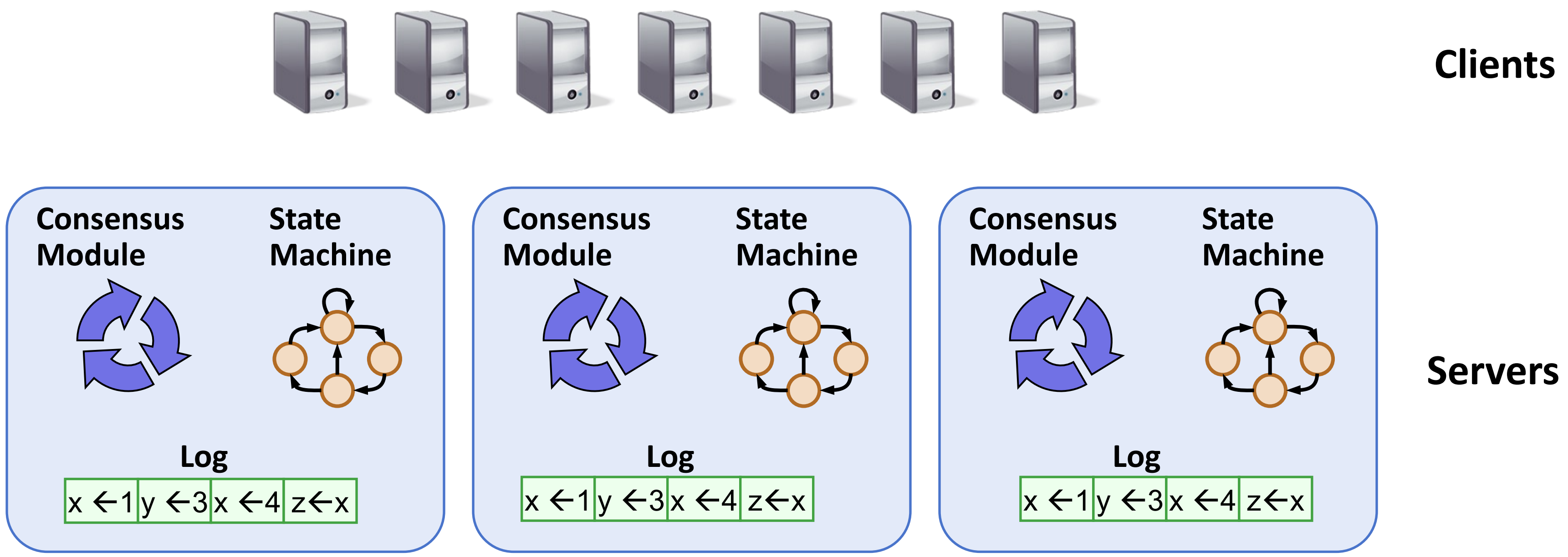
On John Ousterhout

- Highly respected systems researcher

- Inventor of RAMCloud, Tcl scripting language, Sprite Operating System

- Founded multiple successful companies
  - Electric Cloud

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 Protocols):
  - 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

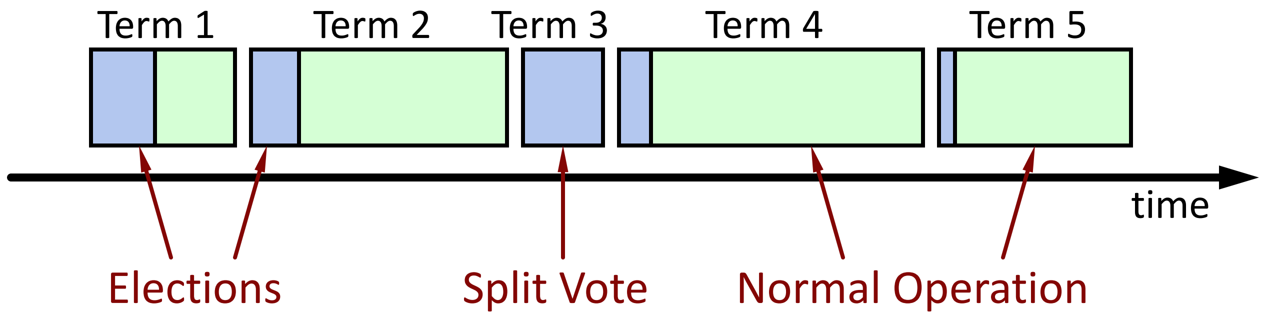
Raft Overview

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

Recap: 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

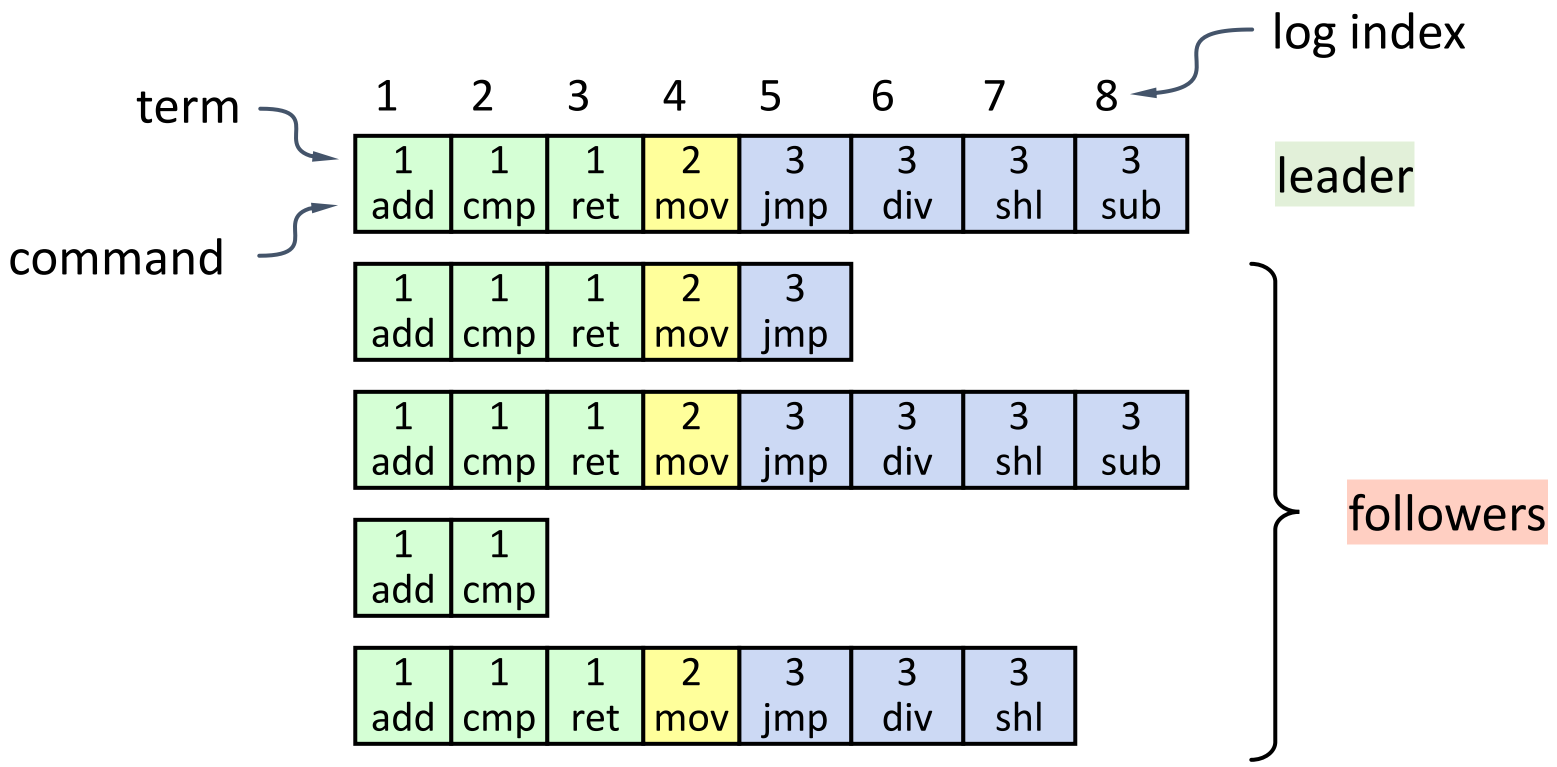
Recap: Terms



- 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

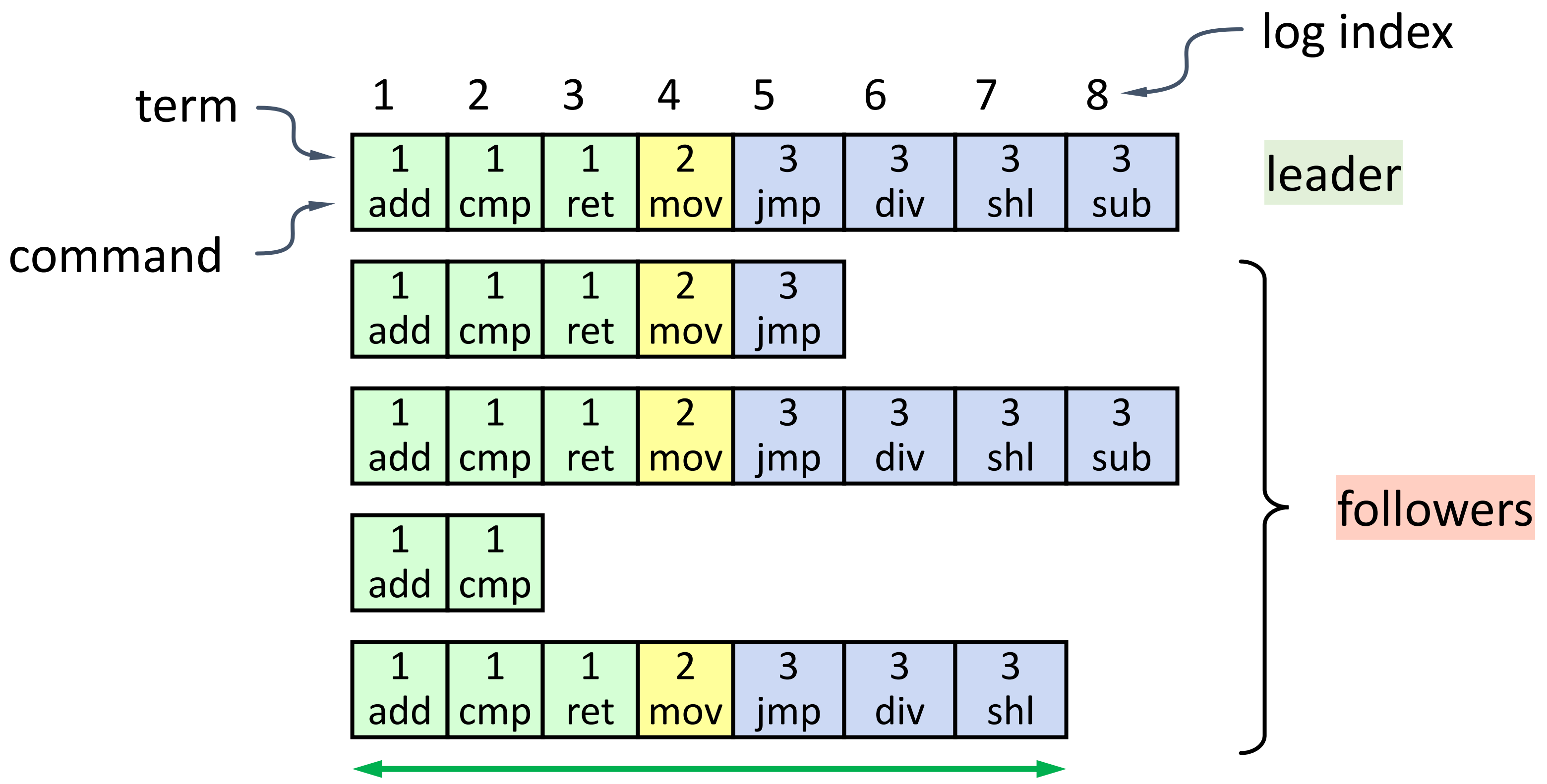
Log Structure



log index
term
command
1 2 3 4 5 6 7 8
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl | 3 sub
leader
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl | 3 sub
1 add | 1 cmp
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div | 3 shl
followers

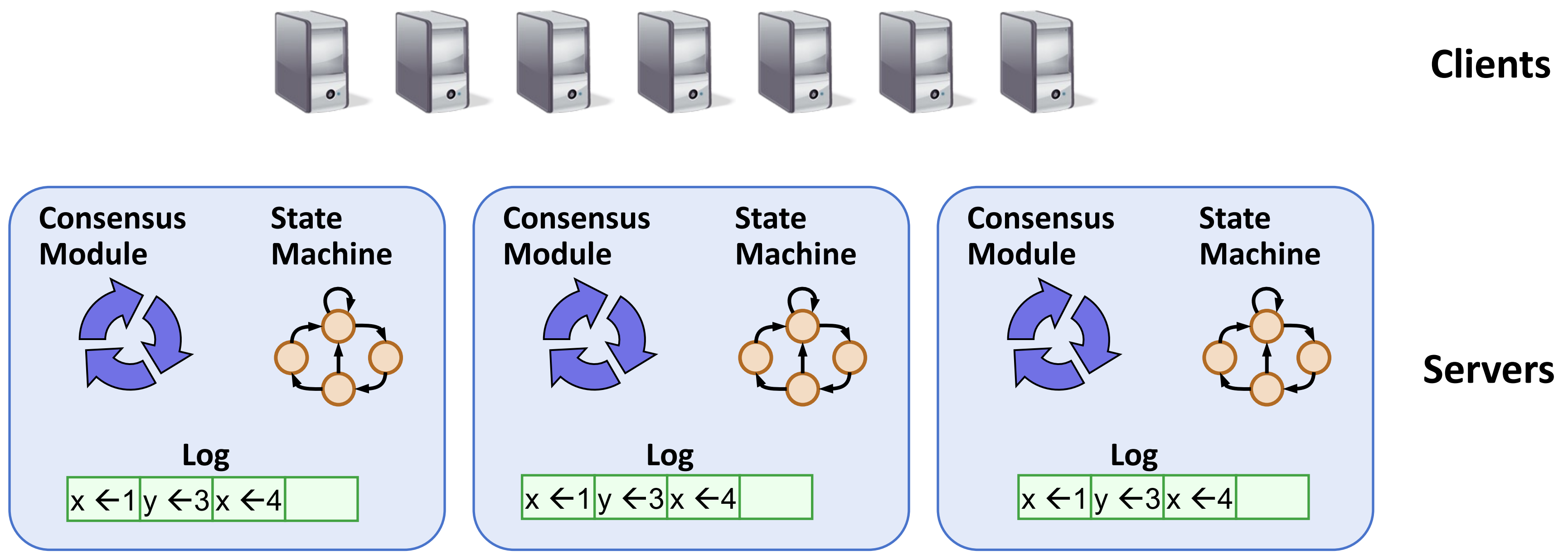
- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes

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



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

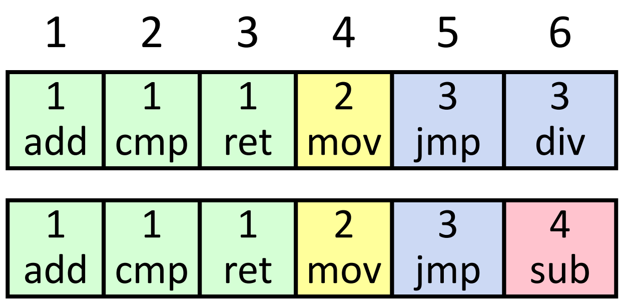
Log Consistency in Raft?

- 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

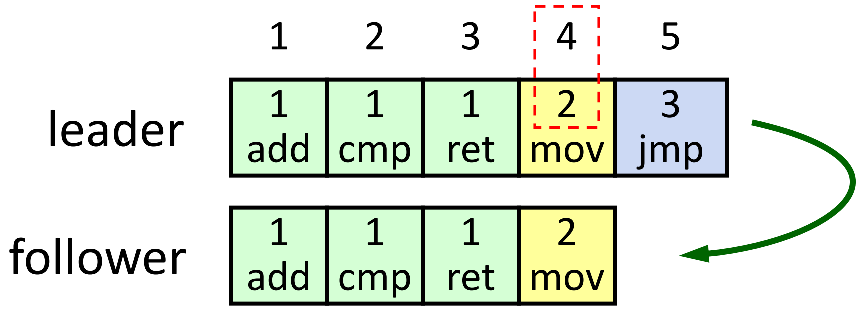

1 2 3 4 5 6
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 3 div
1 add | 1 cmp | 1 ret | 2 mov | 3 jmp | 4 sub

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

How is log consistency ensured?

AppendEntries Consistency Check

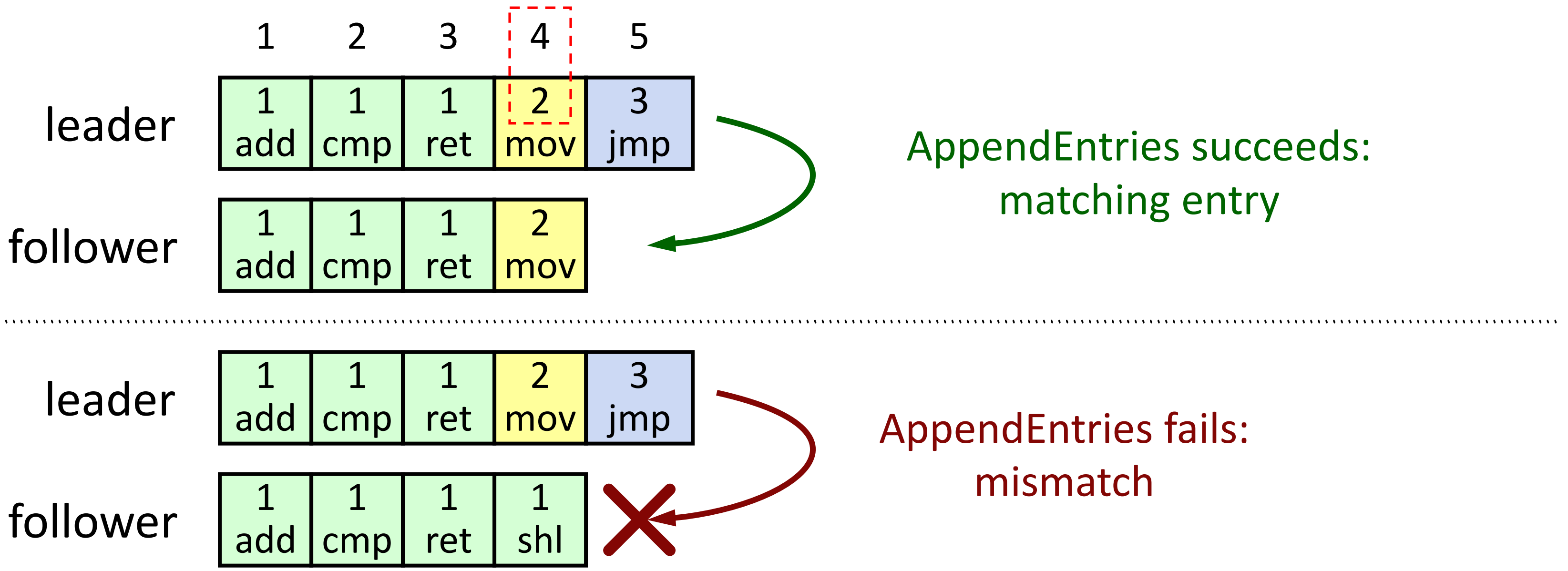
- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request



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

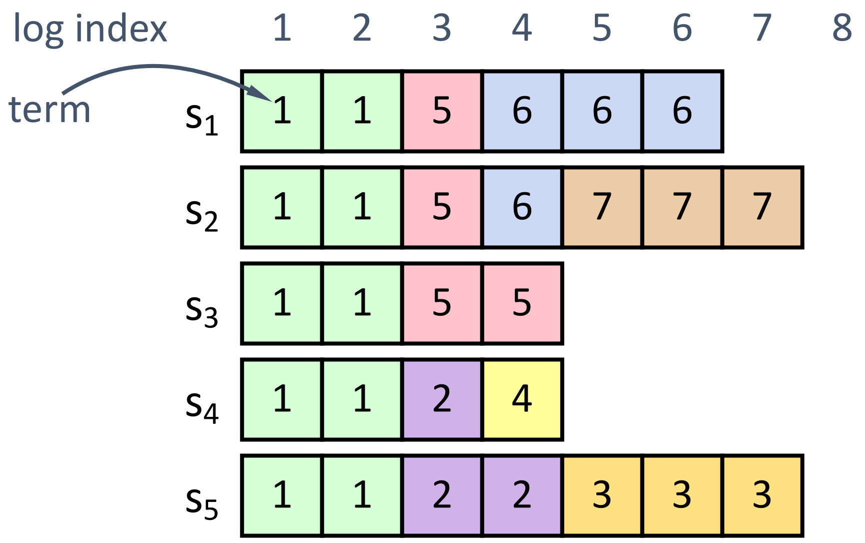
6. Configuration changes:
   - Adding and removing servers

Leader Changes

What could go wrong as the leader changes?

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
  - Multiple crashes can leave many extraneous log entries:



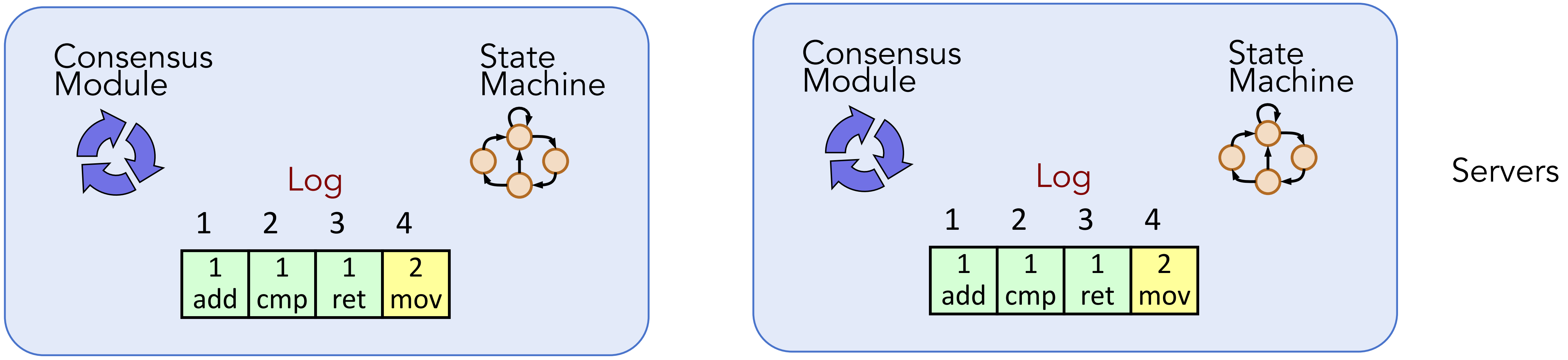
log index	1	2	3	4	5	6	7	8
s_1	1	1	5	6	6	6		
s_2	1	1	5	6	7	7	7	
s_3	1	1	5	5				
s_4	1	1	2	4				
s_5	1	1	2	2	3	3	3	

Safety Requirement

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

Safety Requirement

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



Safety Requirement

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

Safety Requirement

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

$\xrightarrow{}$ Restrictions on leader election

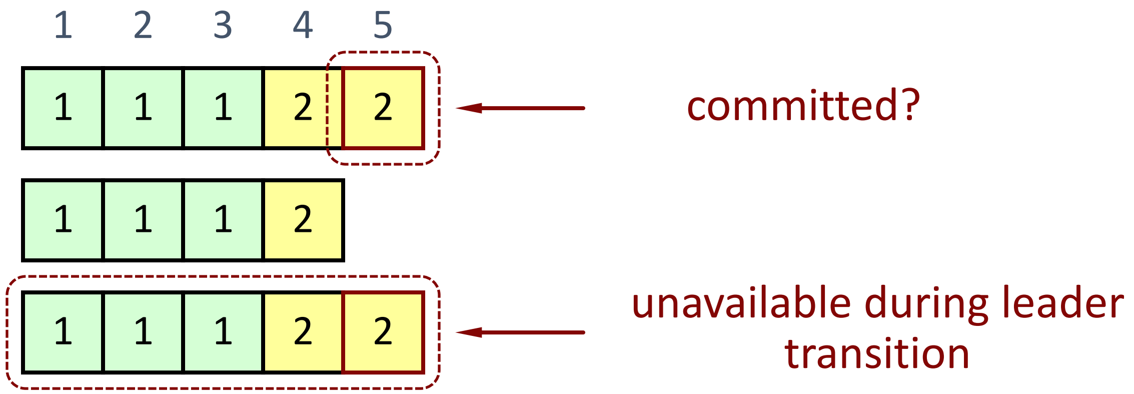
How do we pick the best leader?

Picking the Best Leader

- A new leader should have all the committed entries
- Can we tell which entries are committed?

Picking the Best Leader

- Can’t tell which entries are committed!



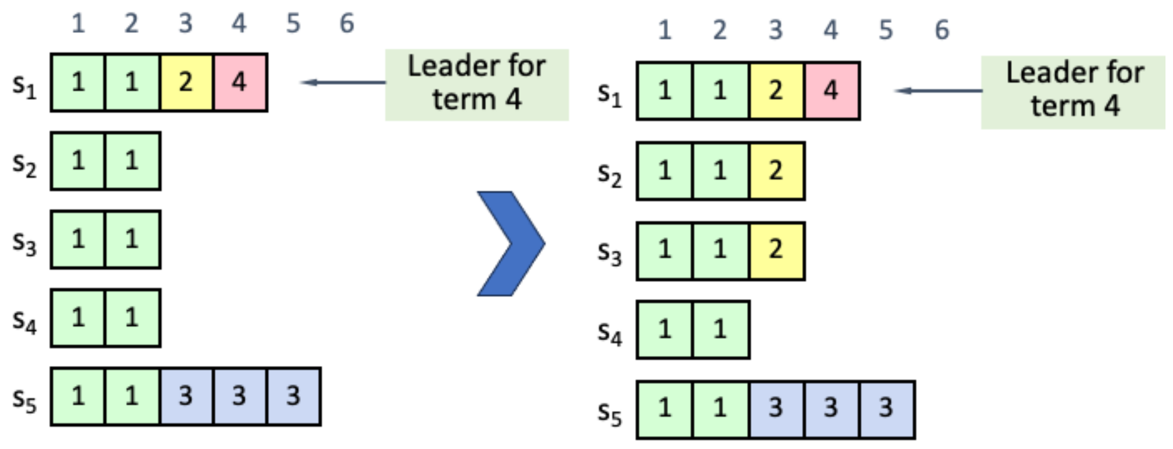
Picking the Best Leader

- Can’t necessarily tell which entries are committed
- 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 complete”:
      $(\text{lastTerm}_V > \text{lastTerm}_C) \mathbin{||}$
      $(\text{lastTerm}_V == \text{lastTerm}_C) \mathbin{\&\&} (\text{lastIndex}_V > \text{lastIndex}_C)$
    - Leader will have “most complete” log among electing majority

Class Exercise

Q1. Which servers can become leaders in this example, if s1 crashes?

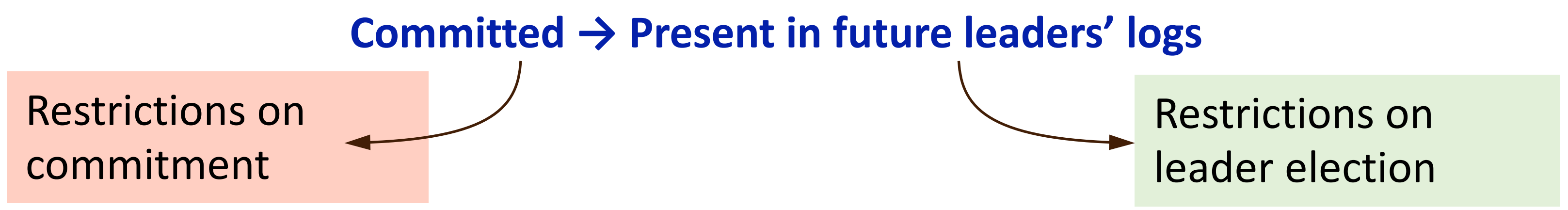
In general, it is possible based on the leader selection criteria in the previous slide that the chosen leader does not have all the committed entries in its log



Safety Requirement

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

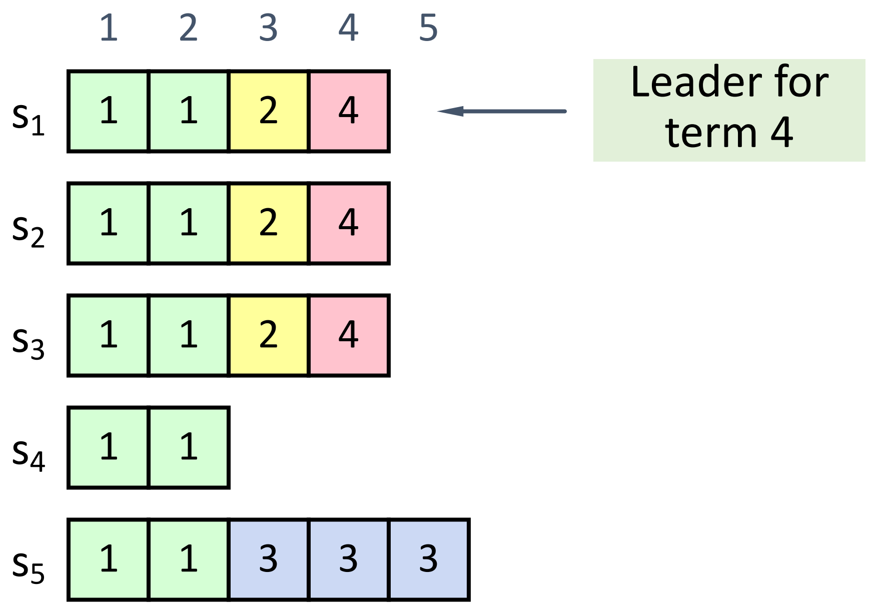
- Raft safety property:
    - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders


Committed → Present in future leaders' logs
Restrictions on commitment
Restrictions on leader election

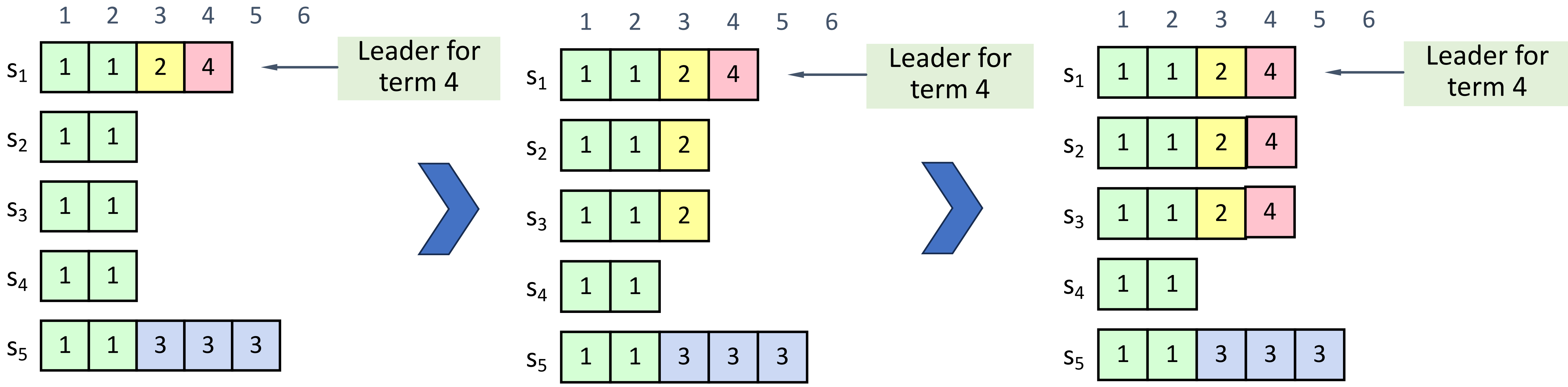
New Commit Rule

New Commitment Rules

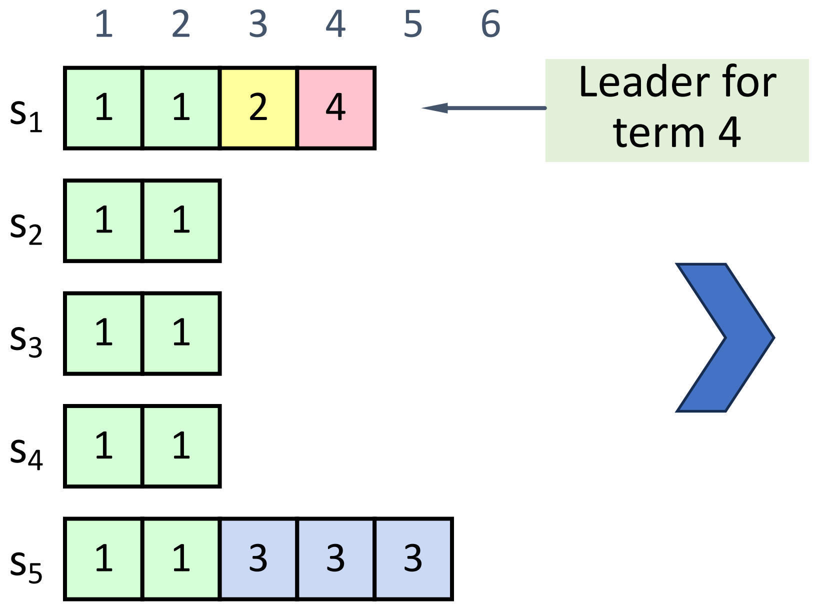
- For a leader to decide an entry is committed:
  1. Must be stored on a majority of servers
  2. At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 stored on majority:
  - S_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe

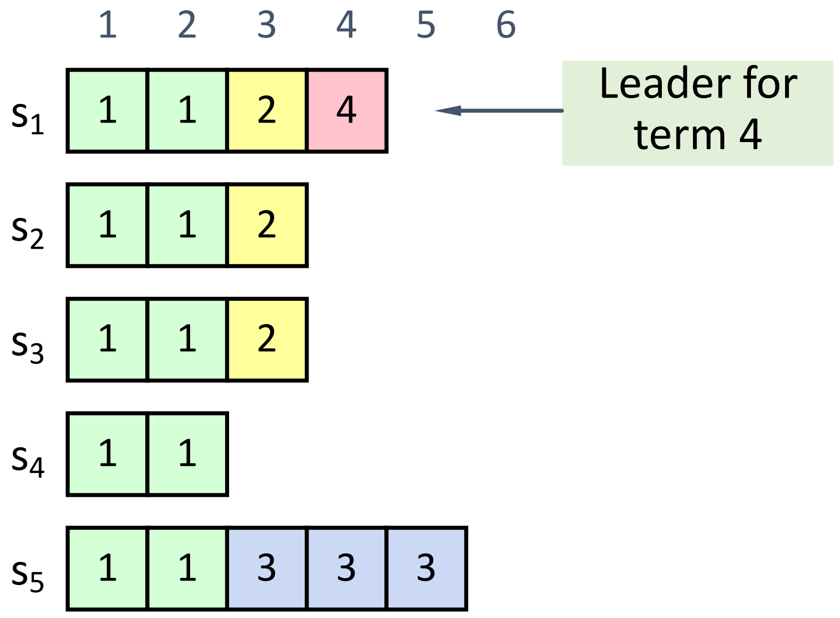


Combination of election rules and commitment rules makes Raft safe


1 2 3 4 5 6
s_1 1 1 2 4
s_2 1 1
s_3 1 1
s_4 1 1
s_5 1 1 3 3 3
Leader for term 4
s_1 1 1 2 4
s_2 1 1 2
s_3 1 1 2
s_4 1 1
s_5 1 1 3 3 3
Leader for term 4
s_1 1 1 2 4
s_2 1 1 2 4
s_3 1 1 2 4
s_4 1 1
s_5 1 1 3 3 3
Leader for term 4

Q: Is it possible for a log entry to be replicated on a majority of servers but to be <u>never</u> executed by the state machines?


1
2
3
4
5
6
s1
s2
s3
s4
s5
Leader for term 4


1
2
3
4
5
6
s1
s2
s3
s4
s5
Leader for term 4

Q: Is it guaranteed that any new leader will have all the committed entries?

Q: Is it guaranteed that any new leader will have the most complete log?

Leader Changes in Raft–Two Key Ideas

- During elections:
  - Voting server V denies vote if its log is "more complete":
    $(\text{lastTerm}_V > \text{lastTerm}_C) \,||$
    $(\text{lastTerm}_V == \text{lastTerm}_C)\ \&\&\ (\text{lastIndex}_V > \text{lastIndex}_C)$
  - Leader will have "most complete" log among electing majority
- New commitment Rule
  - For a leader to decide an entry is committed:

1. Must be stored on a majority of servers &
2. At least one new entry from leader's term must also be stored on majority of servers

Now that we have shown that the leaders log is correct, how do we make the followers log right?

Log Inconsistencies

- Leader changes can result in log incosistencies:

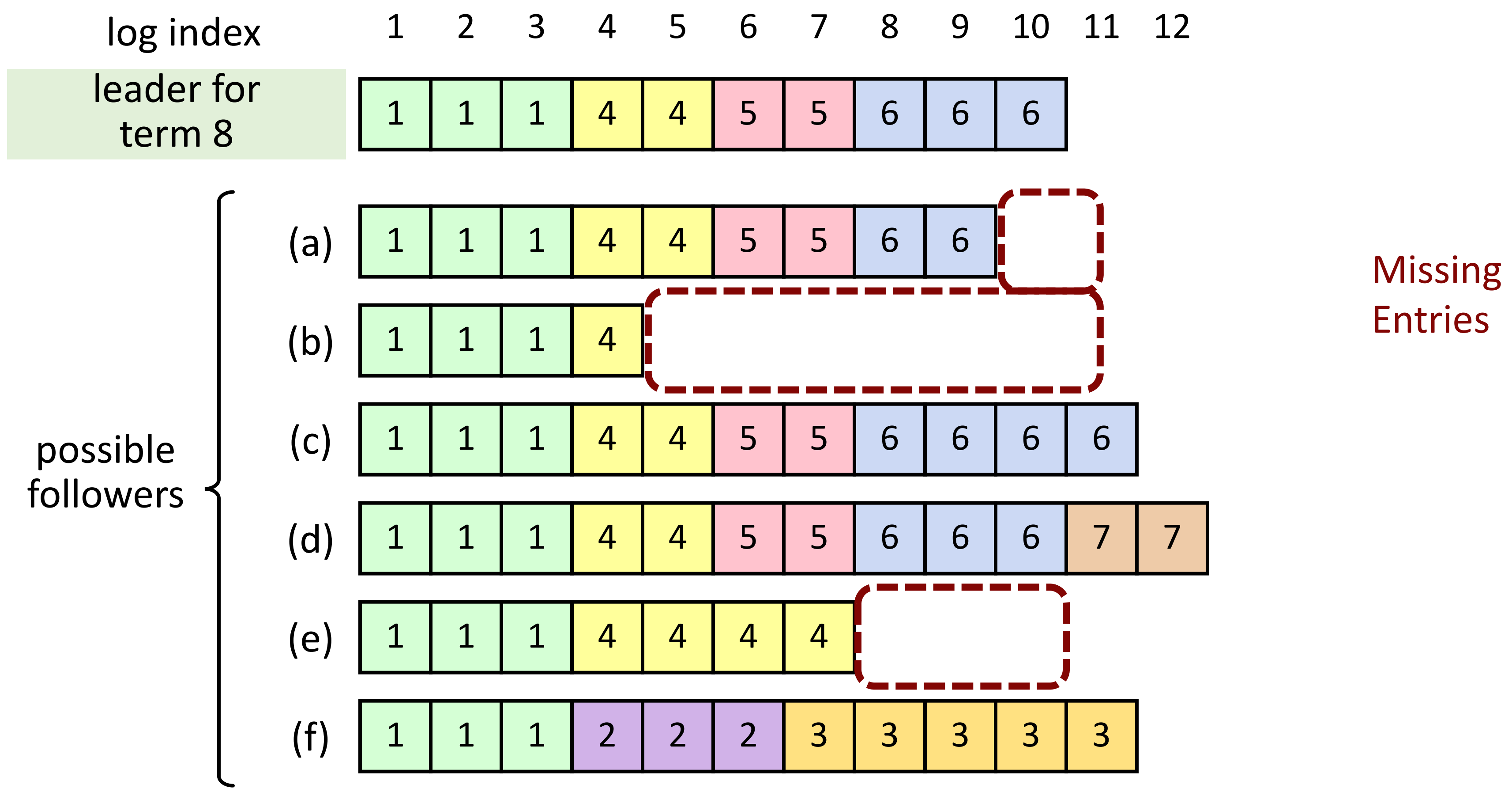
Log Inconsistencies

- Leader changes can result in log incosistencies:

	log index	1	2	3	4	5	6	7	8	9	10	11	12
	leader for term 8	1	1	1	4	4	5	5	6	6	6		
possible followers	(a)	1	1	1	4	4	5	5	6	6			
	(b)	1	1	1	4								
	(c)	1	1	1	4	4	5	5	6	6	6	6	
	(d)	1	1	1	4	4	5	5	6	6	6	7	7
	(e)	1	1	1	4	4	4	4					
	(f)	1	1	1	2	2	2	3	3	3	3	3	

Log Inconsistencies

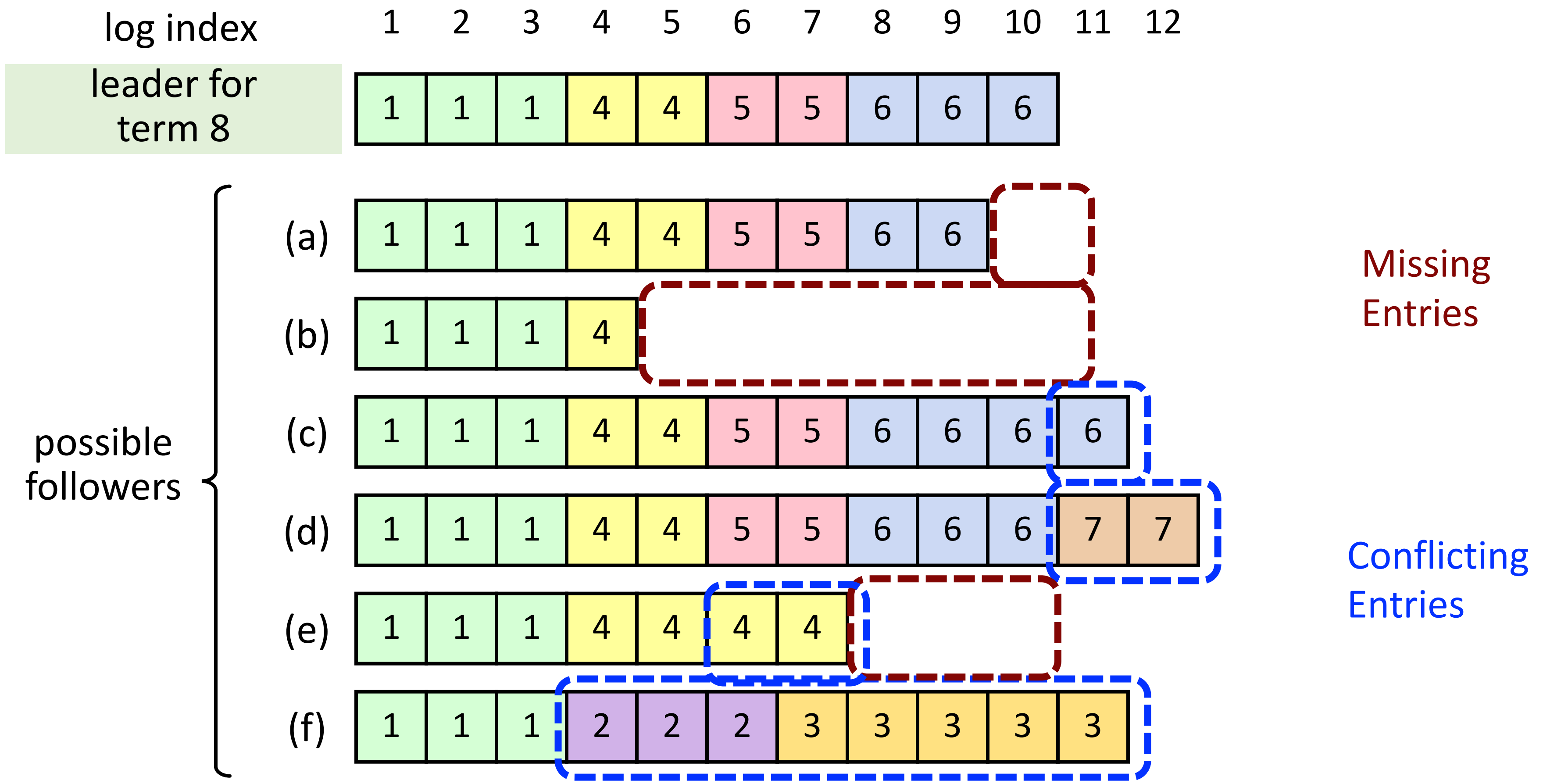
- Leader changes can result in log incosistencies:



log index	1	2	3	4	5	6	7	8	9	10	11	12
leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

Log Inconsistencies

- Leader changes can result in log incosistencies:

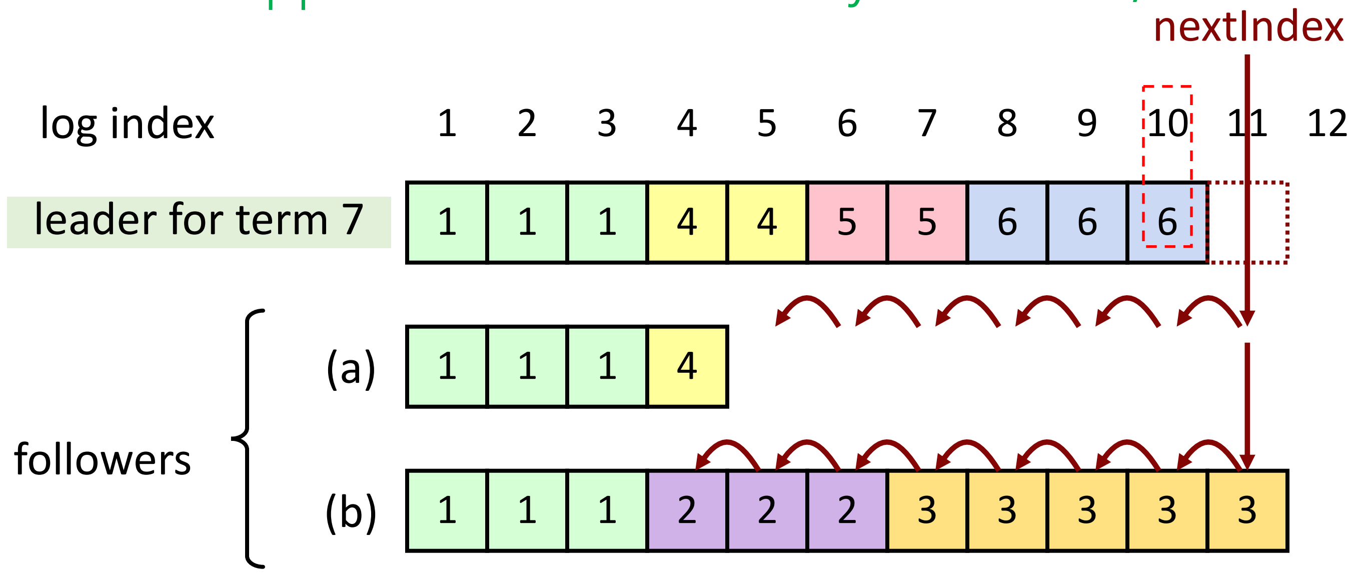


Repairing Follower Logs

- 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 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:

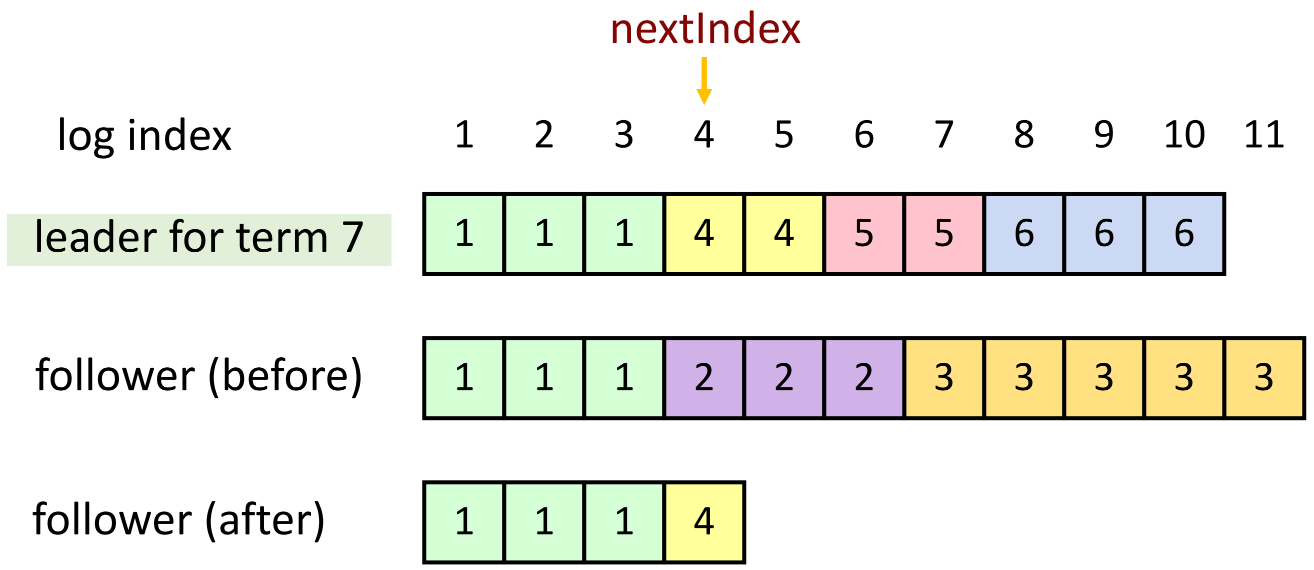


Repairing Follower Logs

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

Repairing Follower Logs

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



Neutralizing Old Leaders

- Deposed leader may not have failed:
  - 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

Client Protocol

- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
- Leader does not respond until command has been logged, committed, and executed by leader's state machine
- If request times out (e.g., leader crash):
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol

- What if leader crashes after executing command, but before responding?
  - Must not execute command twice
- Solution: client embeds a unique id in each command
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command

Raft Summary

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~~
   - ~~Implementing exactly-once semantics~~
6. Configuration changes:
   - Adding and removing servers