Raft: Consistent Log Replication

Jinyang Li

Raft slides are from Ongaro and Ousterhout's raft user study

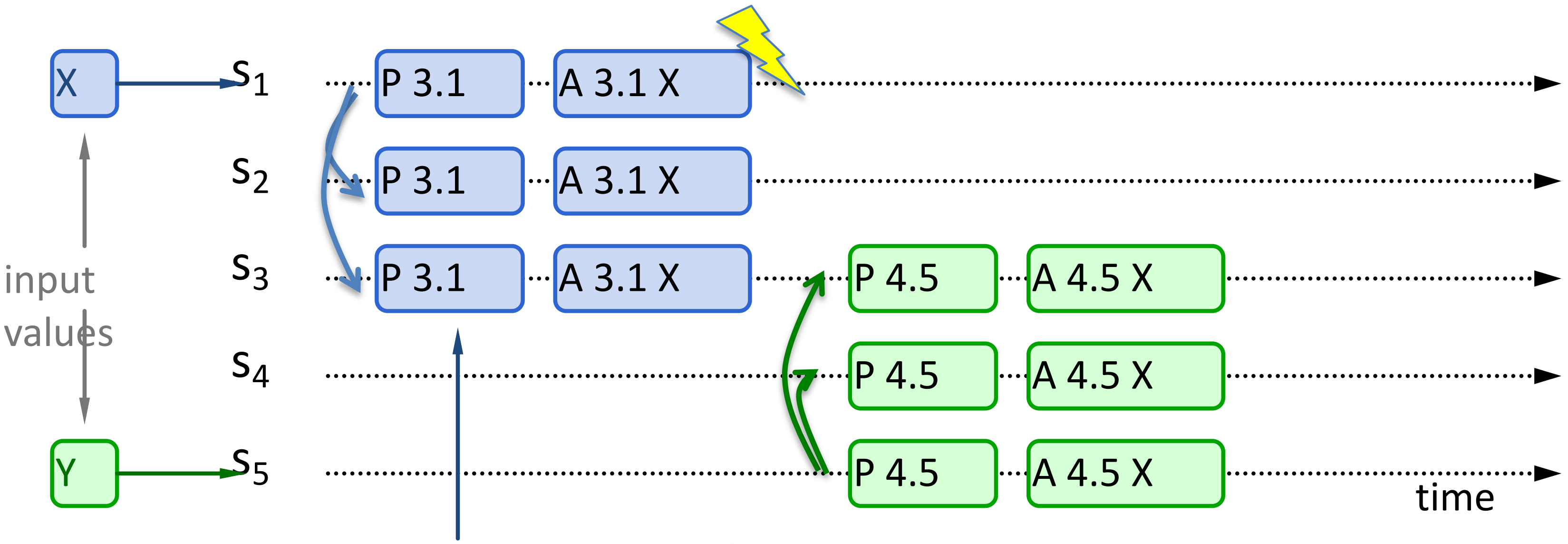
What we’ve learnt last time

- Single-decree Paxos
  - 2f+1 nodes agree on a single value
  - resilient against f crashes.
- MultiPaxos
  - 2f+1 nodes agree on a sequence of values

Recap: Single-decree Paxos

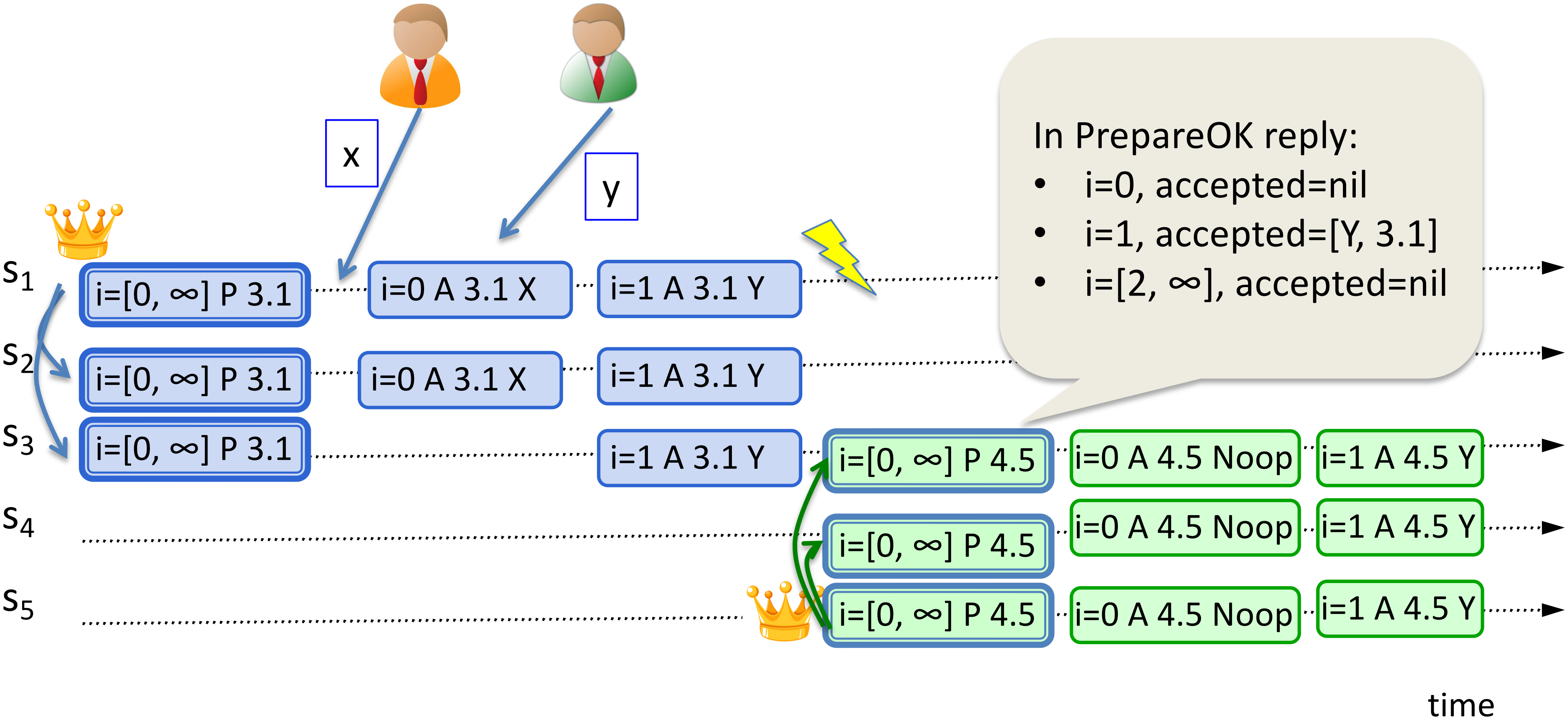
- Paxos invariant (safety property):
  - each proposal has a globally unique number
  - if a proposal p with value v is committed, then all proposal p’ > p has value v
- 2-phase
  - Prepare (phase-1): find a safe value to use for proposal p
    - In accepting Prepare(p), a node
      - returns highest previously accepted proposal
      - promise not to accept any proposal < p in the future
    - Among a majority of OK replies, safe value is:
      - the accepted valued with the highest proposal number
  - Accept (phase-2): make a majority accept proposal p w/ value v
    - If a majority accepts, then p with v is committed

Recap: Single-decree Paxos



"s3 okays Prepare proposal 3.1 (from s_1)"

Recap: MultiPaxos



x

y

s_1 i=[0, ∞] P 3.1 | i=0 A 3.1 X | i=1 A 3.1 Y

s_2 i=[0, ∞] P 3.1 | i=0 A 3.1 X | i=1 A 3.1 Y

s_3 i=[0, ∞] P 3.1 | i=1 A 3.1 Y | i=[0, ∞] P 4.5 | i=0 A 4.5 Noop | i=1 A 4.5 Y

s_4 i=[0, ∞] P 4.5 | i=0 A 4.5 Noop | i=1 A 4.5 Y

s_5 i=[0, ∞] P 4.5 | i=0 A 4.5 Noop | i=1 A 4.5 Y

In PrepareOK reply:

- i=0, accepted=nil
- i=1, accepted=[Y, 3.1]
- i=[2, ∞], accepted=nil

time

- Runs many single-decree Paxos instances
  - i-th instance commits value at i-th position in the sequence

Today: Raft replicated log

- Paxos’ approach (bottom-up)
  - solve single-decree consensus first
  - replicate a sequence of values using single-decree consensus
- Raft’s approach (top-down)
  - directly solve log replication without first solving single-decree consensus

Why learn Raft?

1985
FLP impossibility

1988
Viewstamp
replication

1989
Paxos

2006
[Google]
Chubby lock
service

2010
[Yahoo/Apache]
Zookeeper

2012
[Google]
Spanner
database

2013
Raft

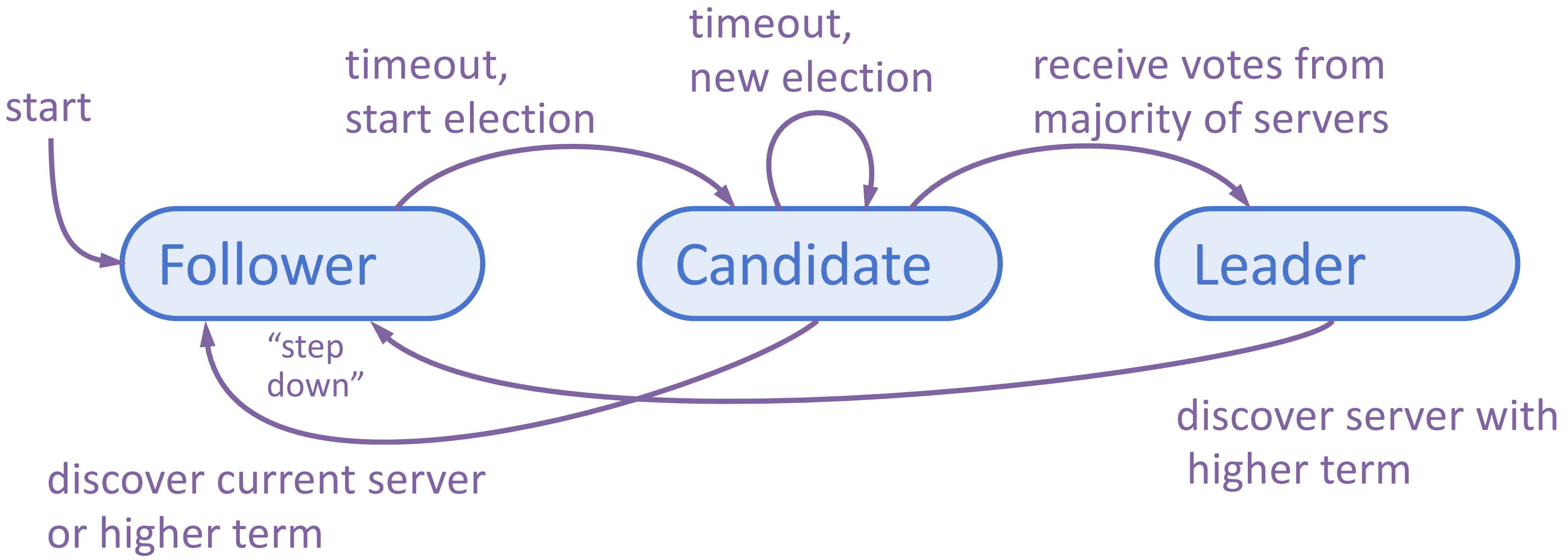
Kubernete/etcd,
RethinkDB,
CockroachDB
YugaByte
TiDB
MongoDB
InfluxDB
…

Raft Overview

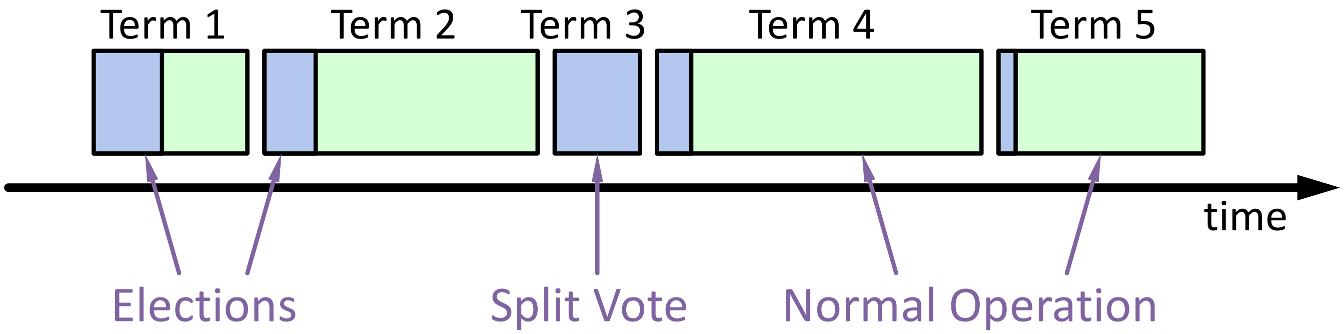
1. Leader election:
   - Select one of the servers to act as leader
2. Normal operation (leader replicates log to others)
3. Safety and consistency
4. Neutralizing old leaders
5. Client interactions
   - Implementing linearizeable semantics
6. Configuration changes:
   - Adding and removing servers

Overview: Raft 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: passive (only responds to incoming RPCs)
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, others are followers



Terms



- Time divided into terms:
  - Each term starts with an election
  - Ends with one leader or no leader
- Each leader is uniquely associated with a term
- Each server maintains current term value
- Key role of terms: identify obsolete information

Raft Protocol Summary

Persistant state

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

Active role: candidates/leader

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

Passive role: followers

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

Heartbeats and Timeouts

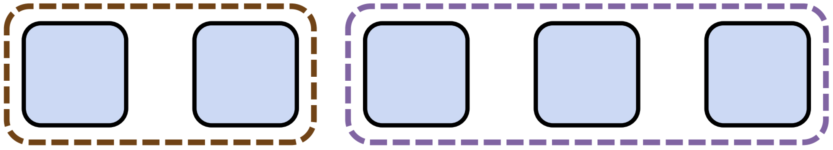
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Election Basics

- Change to candidate state
- Increment current term
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
  2. Receive RPC from a valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

Elections, cont'd

- Safety: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - ⚠ Different candidates may use the same term
    - Node keeps a votedFor variable to ensure it only gives vote to one



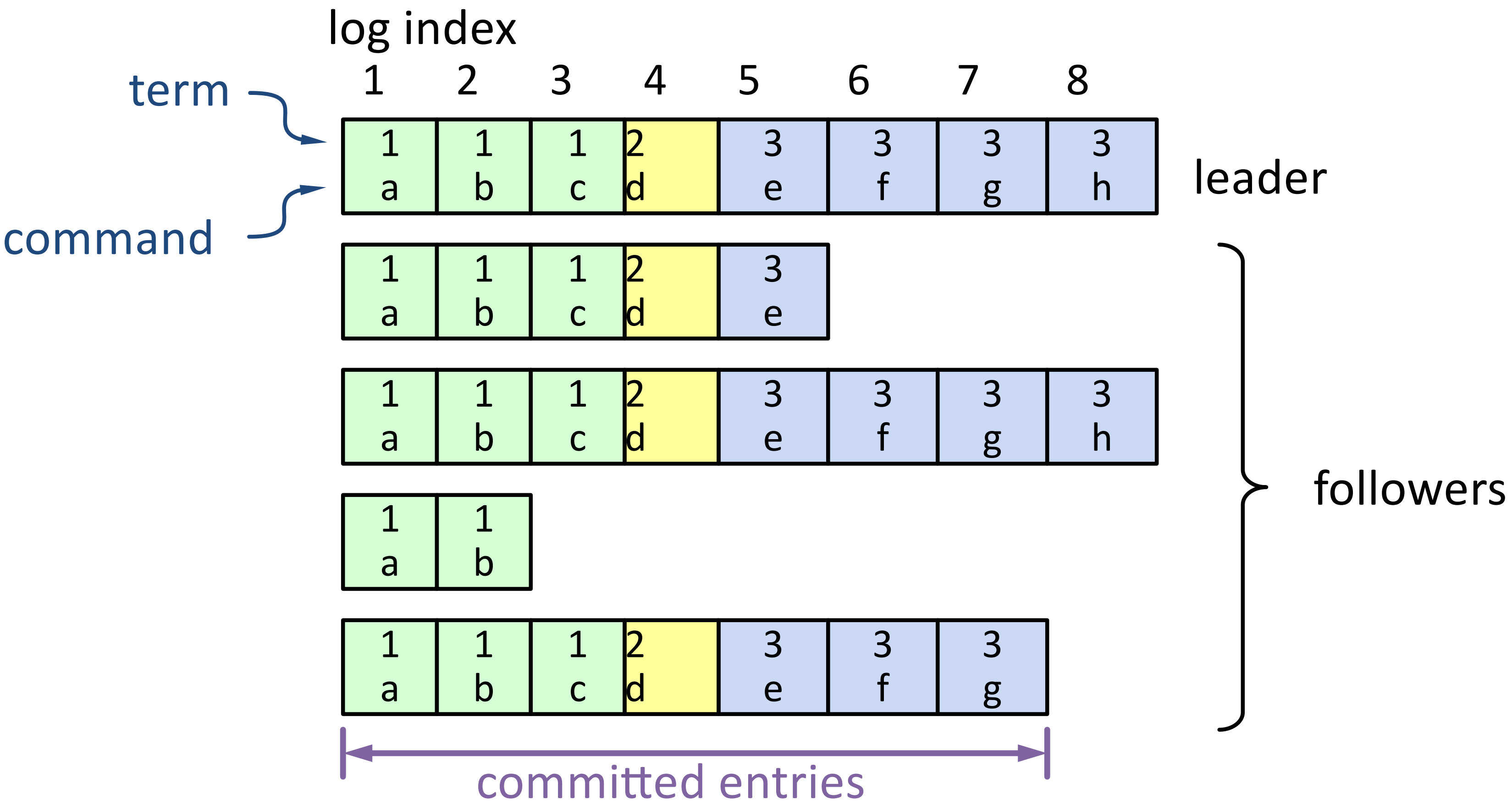
B can't also get majority

Voted for candidate A

Servers

- Liveness: some candidate must eventually win
  - Wait for a randomized amount of time before each retry
  - One server usually times out and wins election before others wake up

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
  - Durable, will eventually be executed by state machines

Normal Operation

- 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 for execution

Log Consistency

Raft tries to achieve the following properties for its logs:

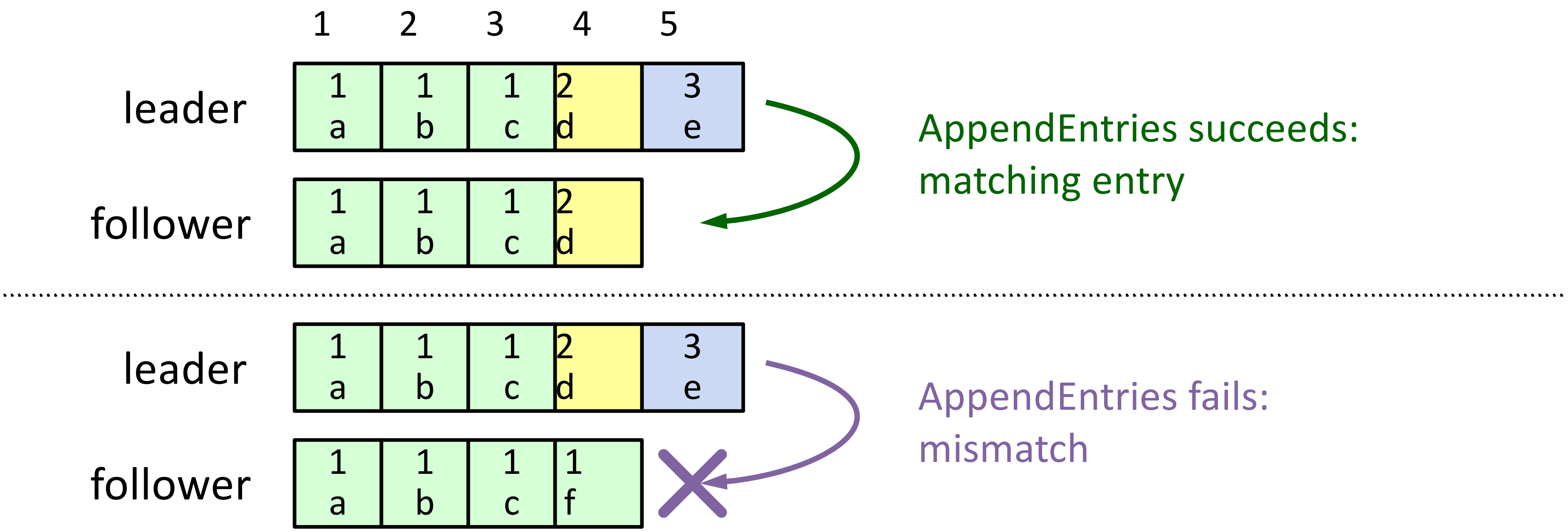
1. If log entries on different servers have same index and term:
   - They store the same command
   - The logs are identical in all preceding entries

1	2	3	4	5	6
1 a	1 b	1 c	2 d	3 e	3 f
1 a	1 b	1 c	2 d	3 e	4 g

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

AppendEntries Consistency Check

- AppendEntries RPC contains <index, term> of the entry e that precedes the new one(s)
- Follower must contain the matching entry e;
  - otherwise it rejects request
- This check ensures log consistency

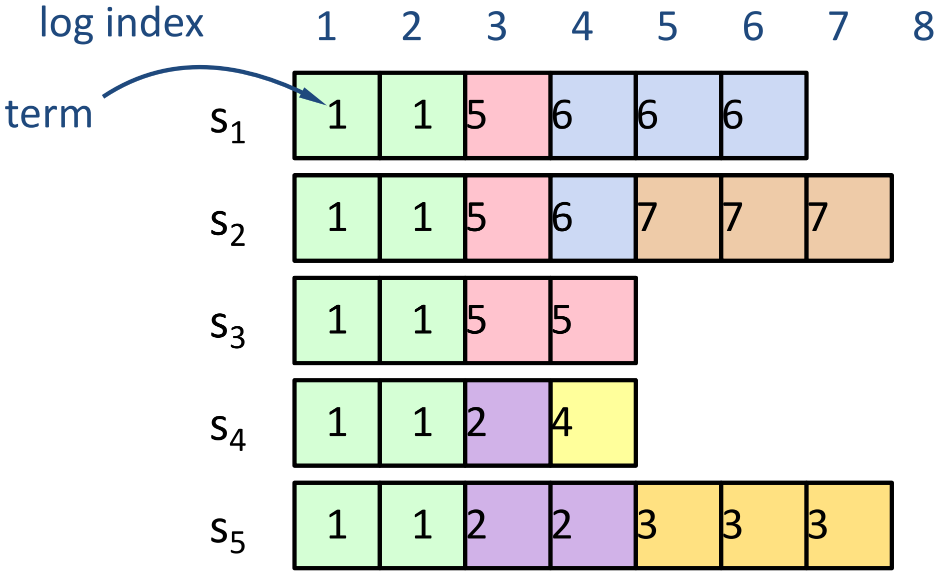


AppendEntries succeeds: matching entry

AppendEntries fails: mismatch

Leader Changes

- New leader's log is "the truth"
- Make followers' logs eventually identical to leader's
- Followers may need to "roll back"
  - old leader may have left entries partially replicated



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

Safety property: no two servers can commit different commands at the same log index

1. Each term has one elected leader
2. If the leader for term t has committed command v at index i, then all leaders for term t’ > t has command v at log index i (and thus will have v committed at i too)

Committed → Present in future leaders’ logs

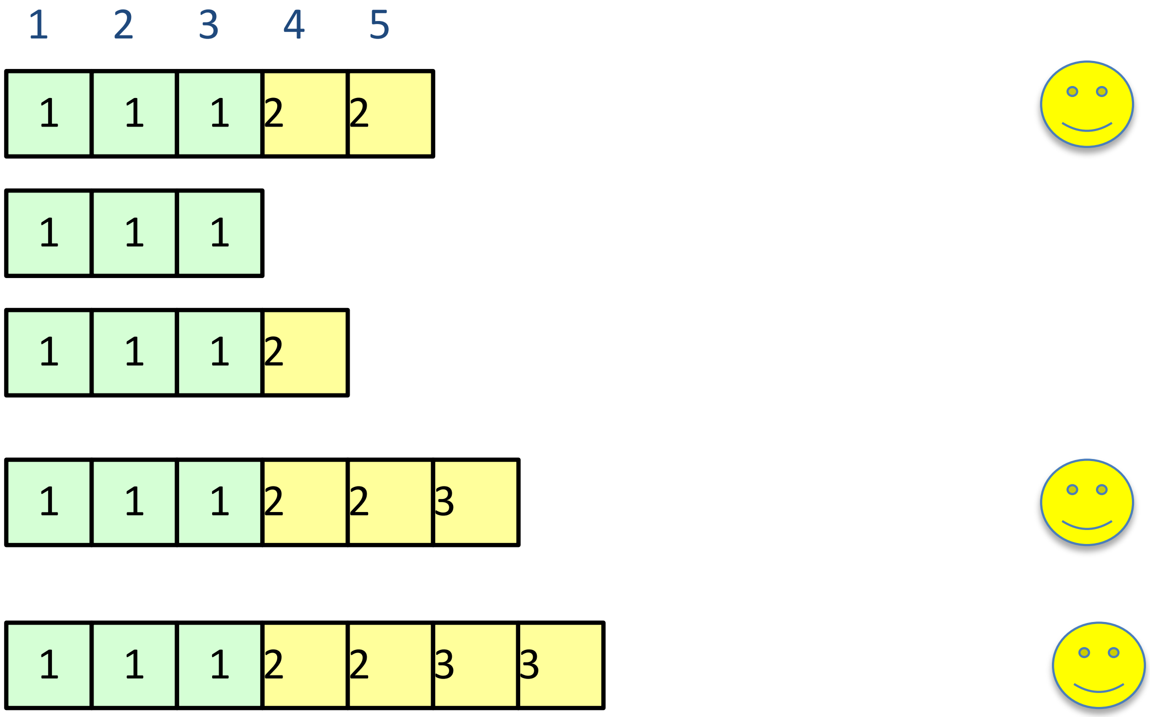
Restrictions on commitment

Restrictions on leader election

New leader must use a safe log

- A safe log is one that’s guaranteed to contain all previously committed commands
- A safe log can be found among a majority quorum of logs according to 2 rules:
  - Rule #1: It is the log with the unique highest term
  - Rule #2: If there are >1 log with the same highest term, it is the longest log among those.

Safe log



1	2	3	4	5			
1	1	1	2	2			☺
1	1	1					
1	1	1	2				
1	1	1	2	2	3		☺
1	1	1	2	2	3	3	☺

Only nodes with a safe log can be elected leader

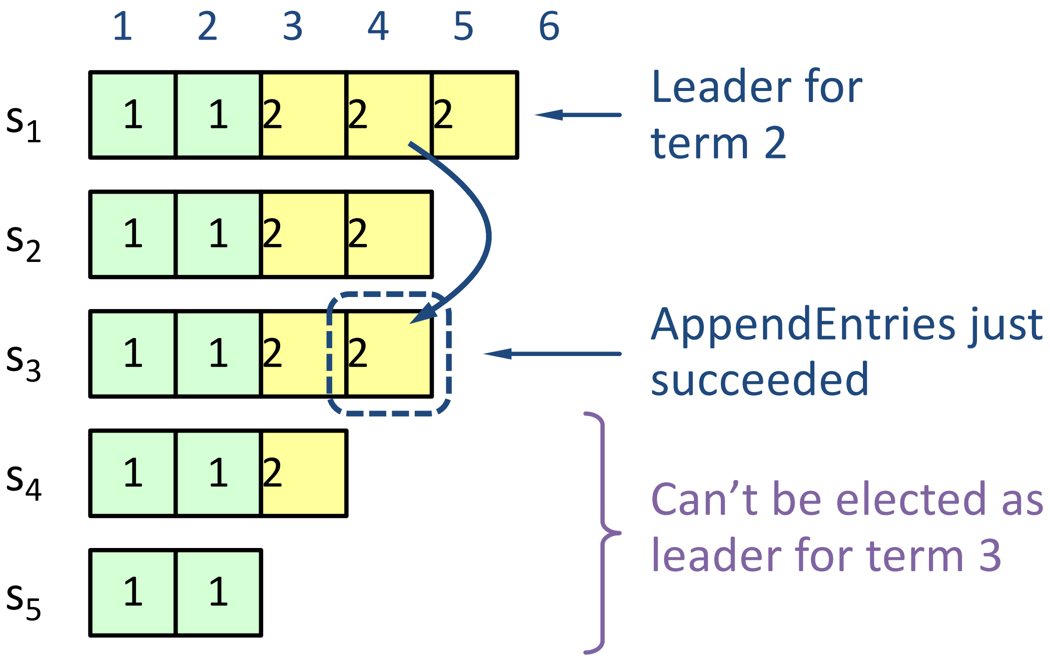
- Instead of transferring logs to leader, Raft ensures that only nodes with a safe log can be elected as leader
- Candidates include log info in RequestVote RPCs (index & term of last log entry)
- Voting server V denies vote if its log is “safer”:
  $(\text{lastTerm}_V > \text{lastTerm}_C)$ ||
  $(\text{lastTerm}_V == \text{lastTerm}_C)$ && $(\text{lastIndex}_V > \text{lastIndex}_C)$
- Leader will have a safe log among electing majority

The subtle caveat of Raft (Sec 5.4.2)

- A log entry’s term does not change since it’s first written
- Can Raft considers an entry committed if majority AppendEntries succeed?

Committing Entry from Current Term

- Case #1/2: Leader decides entry in current term is committed



Leader for term 2

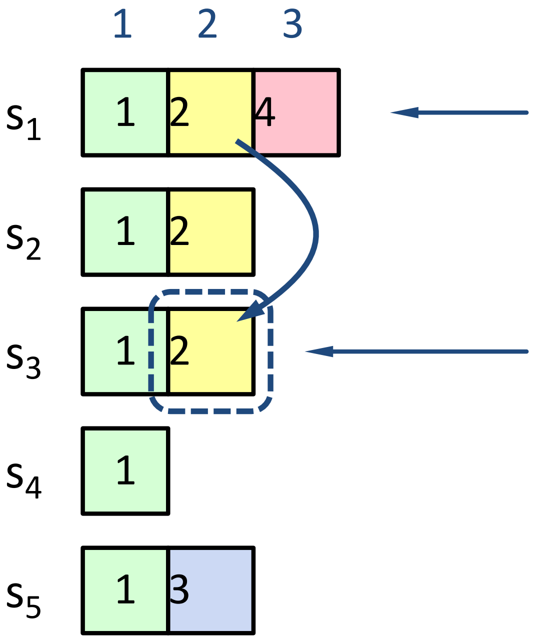
AppendEntries just succeeded

Can’t be elected as leader for term 3

- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

- Case #2/2: Leader is trying to finish committing entry from an earlier term



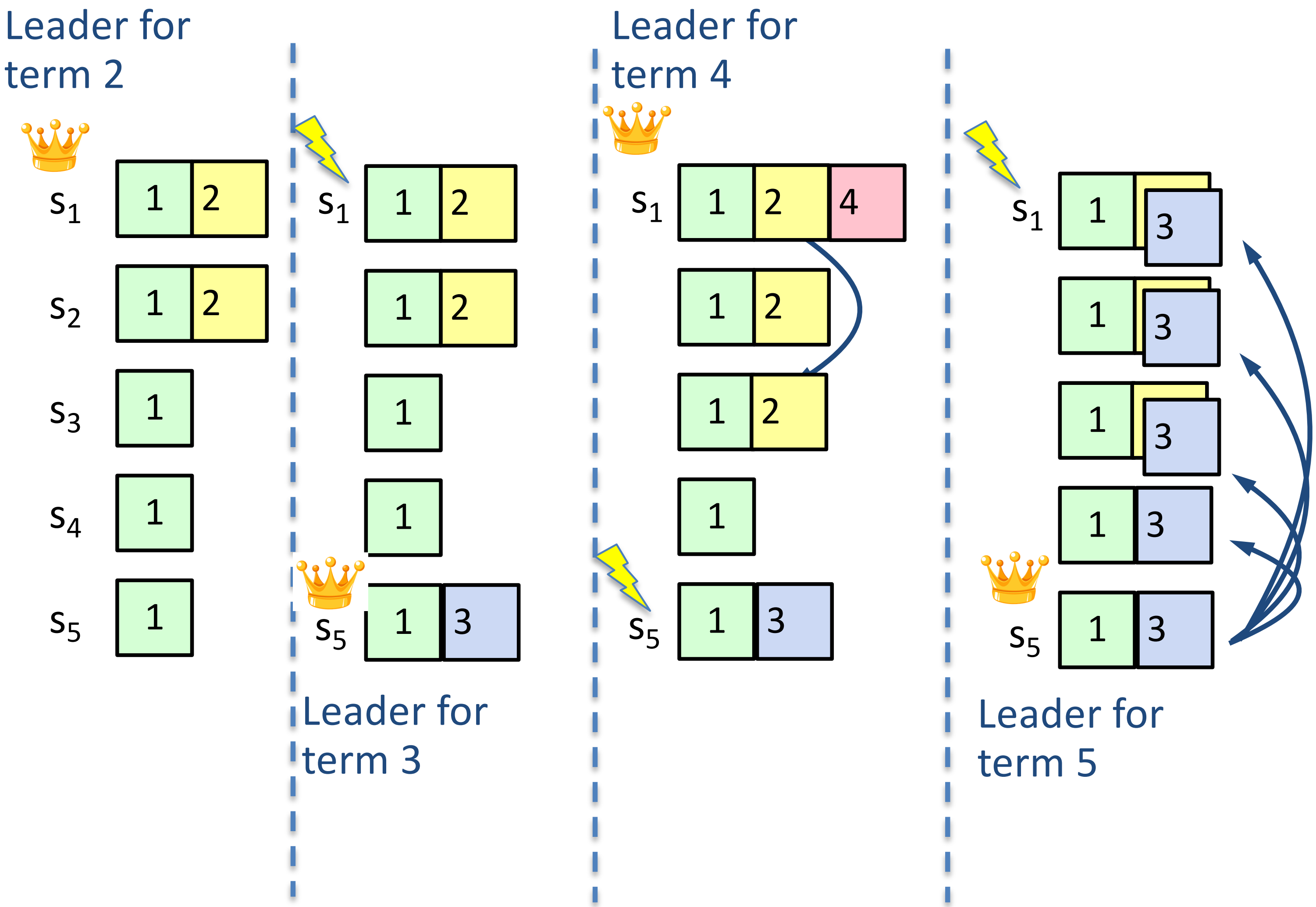
Leader for term 4

AppendEntries just succeeded

- Entry 2 not safely committed:
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 2 on s_1, s_2, and s_3!

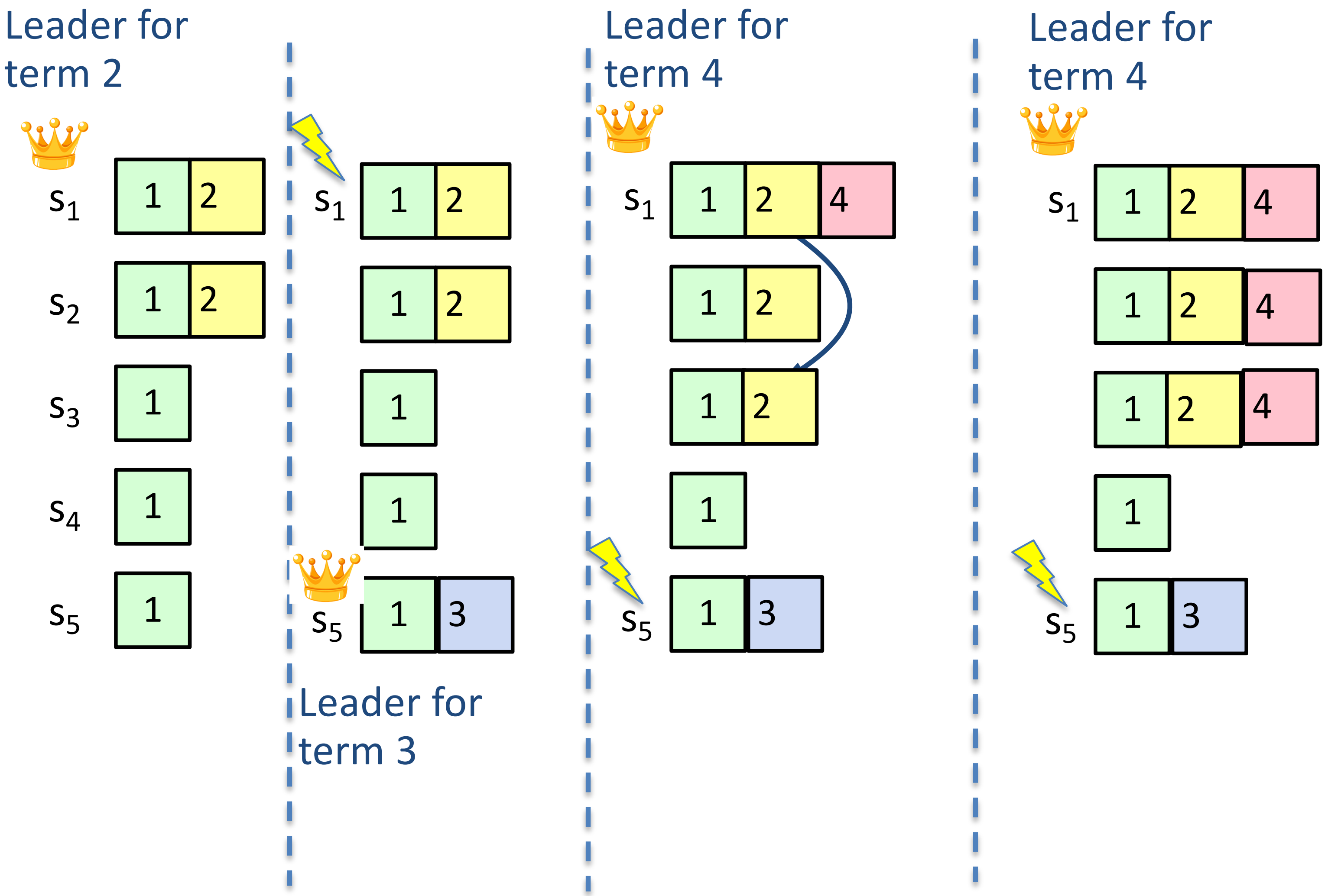
Committing Entry from Earlier Term

- Case #2/2: Leader is trying to finish committing entry from an earlier term



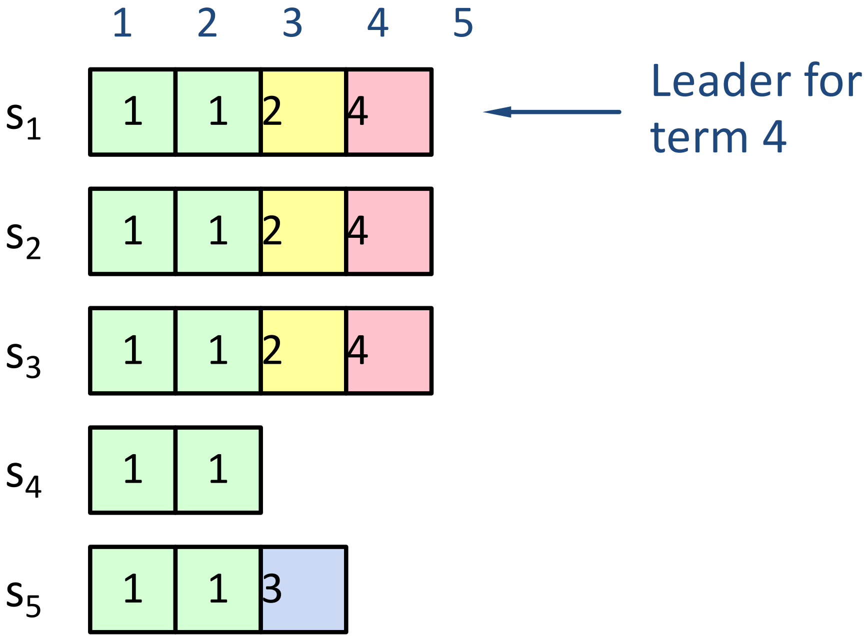
Committing Entry from Earlier Term

- Case #2/2: Leader is trying to finish committing entry from an earlier term



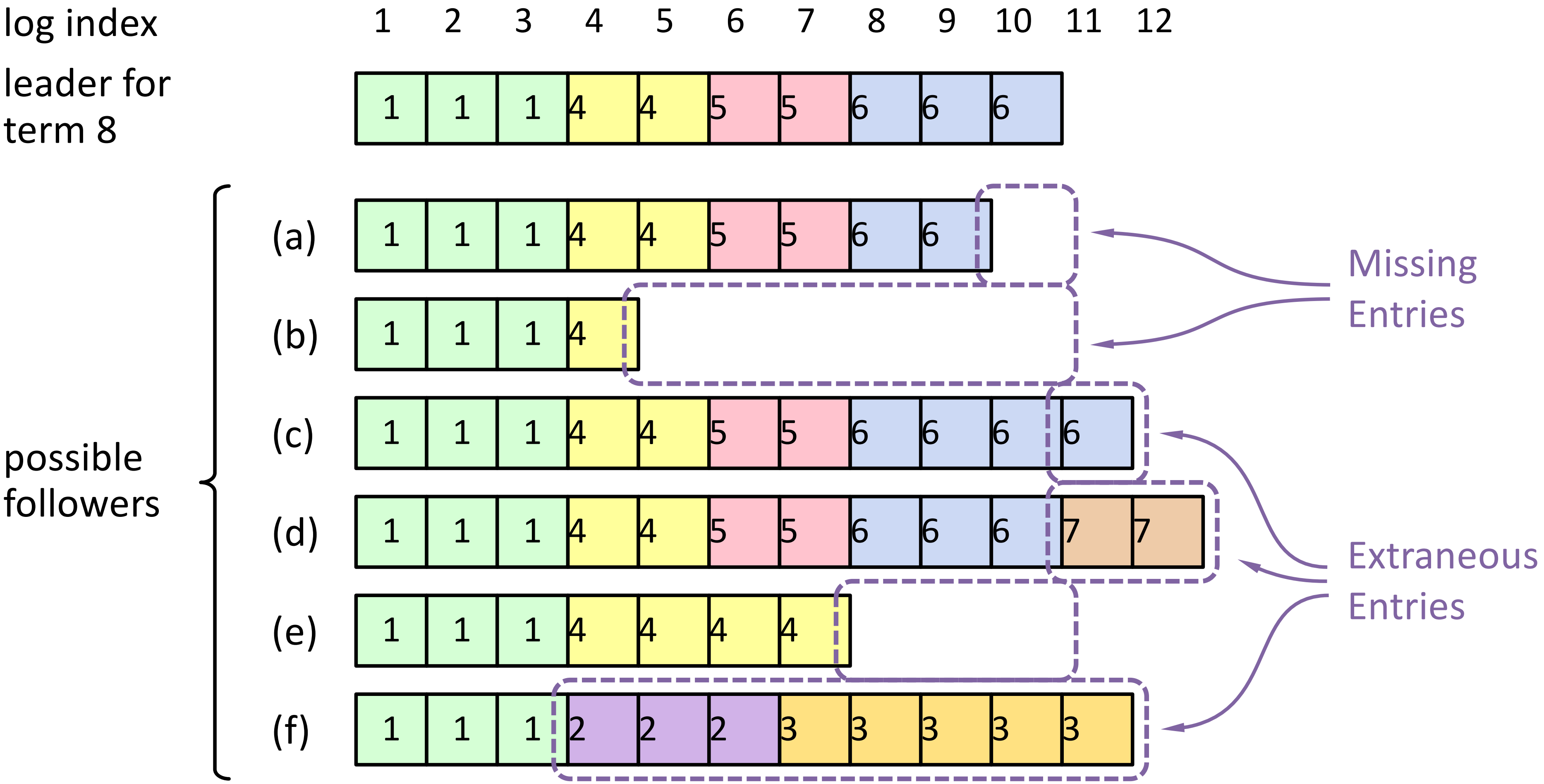
New Commitment Rules

- For a leader to consider an entry as committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's 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



Combination of election rules and commitment rules makes Raft safe

Synchronizing followers' log with leader's log



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	

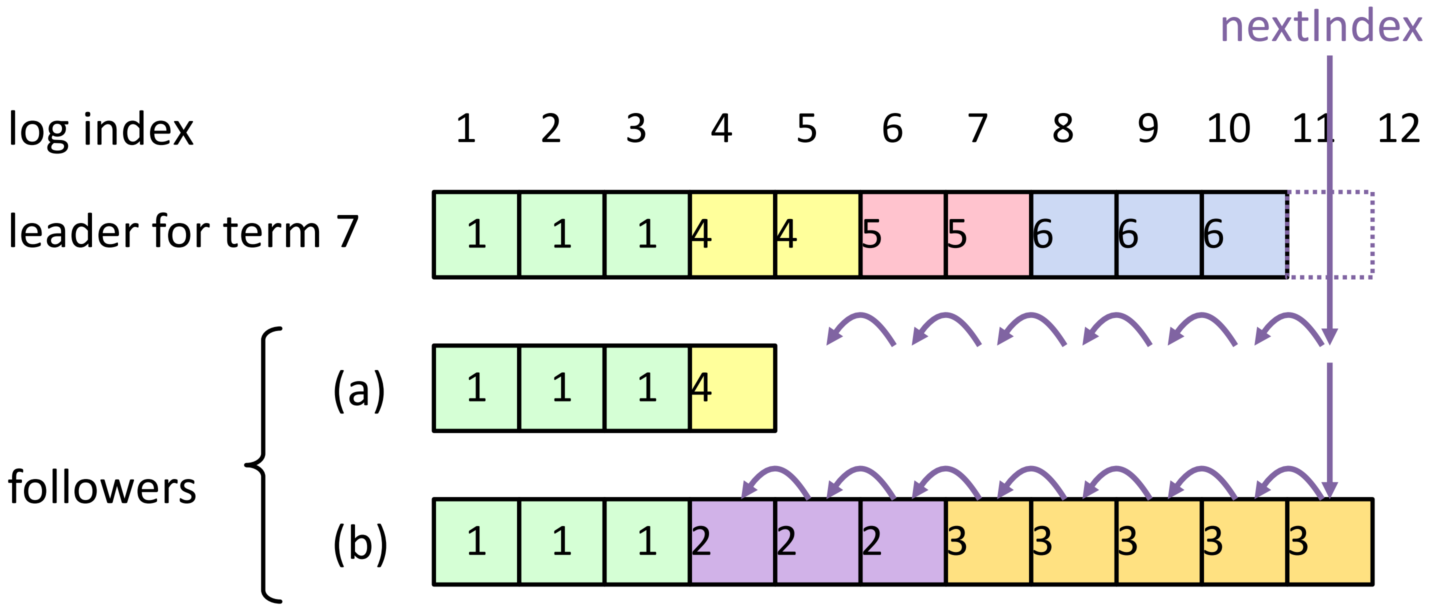
possible followers

Missing Entries

Extraneous Entries

Synchronizing Follower Logs

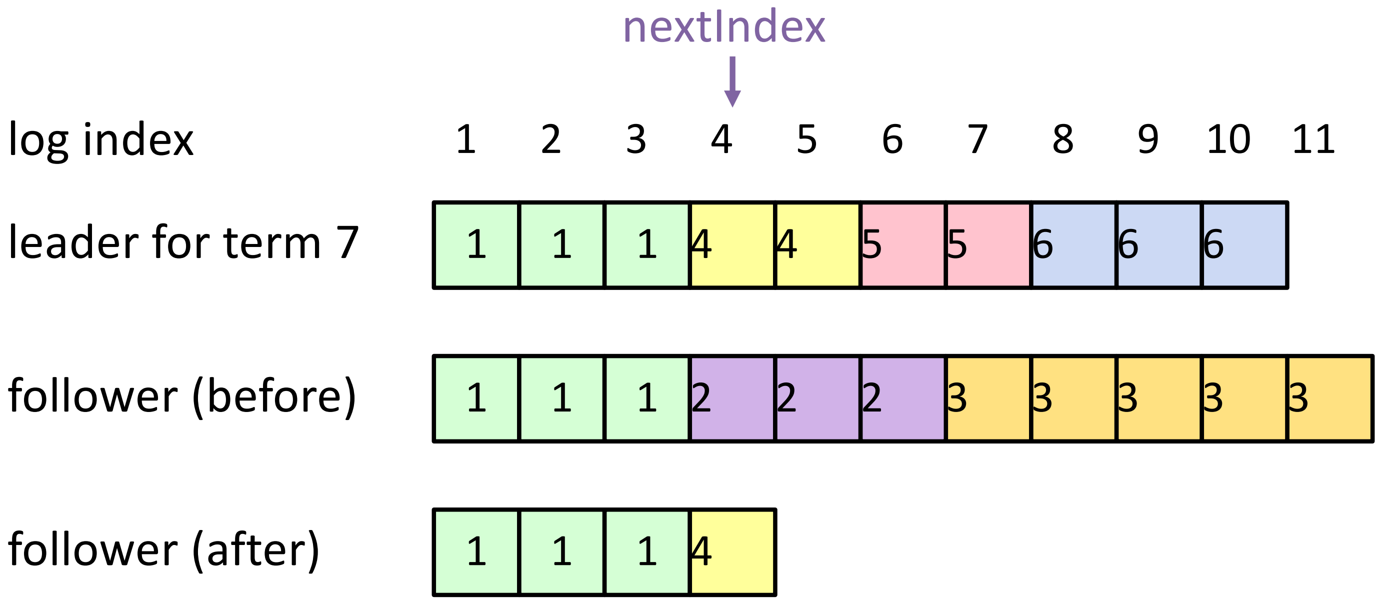
- New leader must make a follower’s log consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader’s last index)
- When AppendEntries check fails, decrement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



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

Repairing Logs, cont’d

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



nextIndex

log index	1	2	3	4	5	6	7	8	9	10	11
leader for term 7	1	1	1	4	4	5	5	6	6	6	
follower (before)	1	1	1	2	2	2	3	3	3	3	3
follower (after)	1	1	1	4							

Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader reconnects and 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

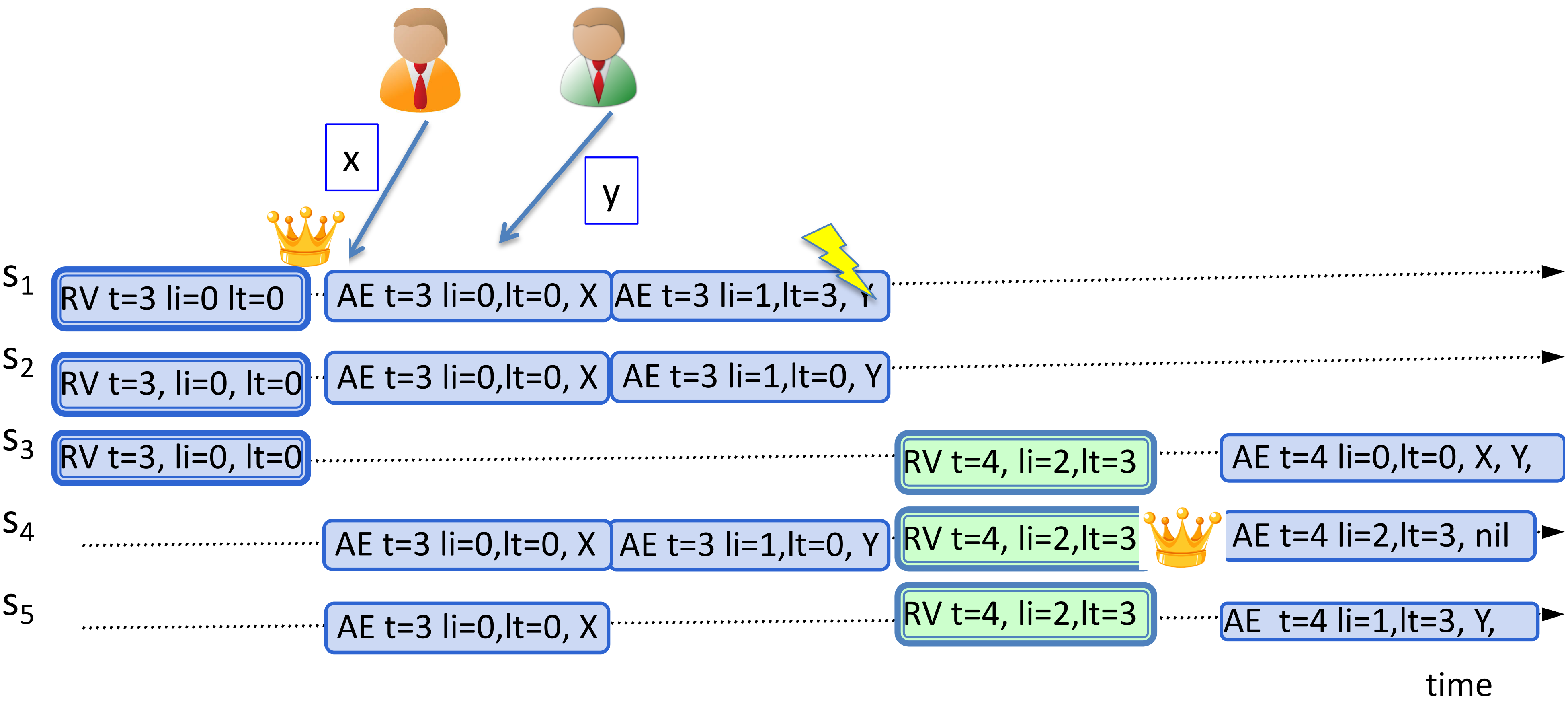
Client Protocol

- 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 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, cont’d

- 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
- Result: exactly-once semantics as long as client doesn’t crash

Recap: Raft



x

y

S_1 RV t=3 li=0 lt=0 · AE t=3 li=0,lt=0, X · AE t=3 li=1,lt=3, Y

S_2 RV t=3, li=0, lt=0 · AE t=3 li=0,lt=0, X · AE t=3 li=1,lt=0, Y

S_3 RV t=3, li=0, lt=0 · RV t=4, li=2,lt=3 · AE t=4 li=0,lt=0, X, Y,

S_4 AE t=3 li=0,lt=0, X · AE t=3 li=1,lt=0, Y · RV t=4, li=2,lt=3 · AE t=4 li=2,lt=3, nil

S_5 AE t=3 li=0,lt=0, X · RV t=4, li=2,lt=3 · AE t=4 li=1,lt=3, Y,

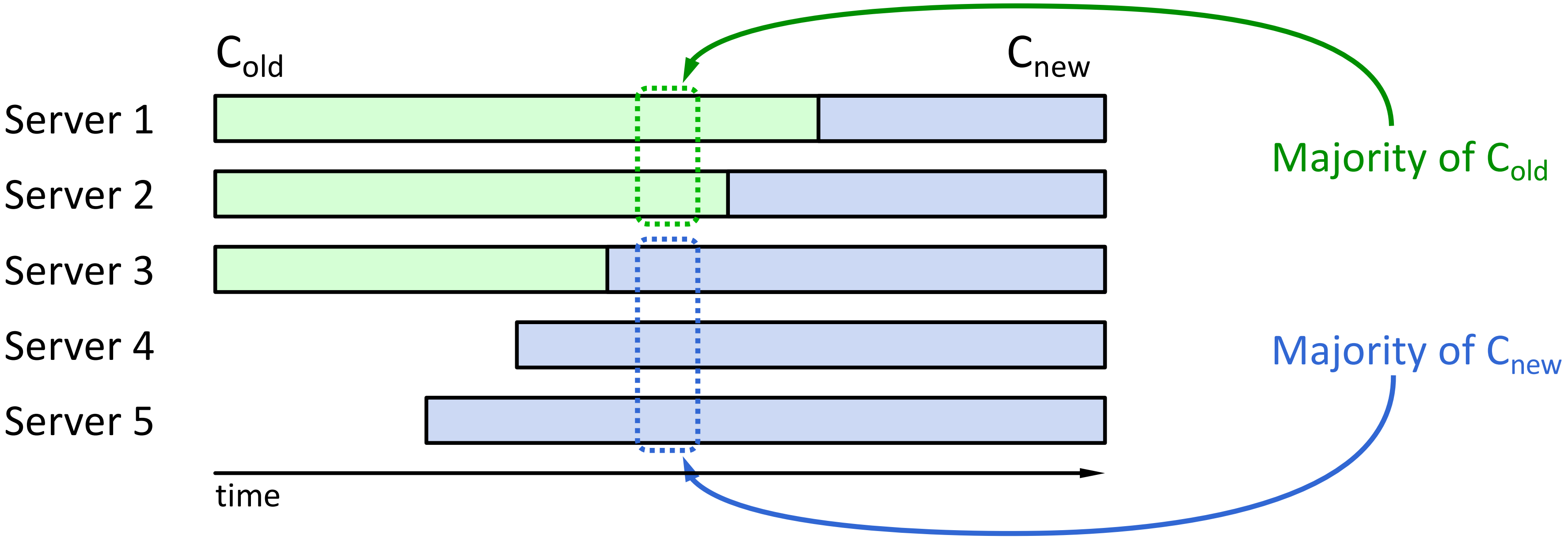
time

Configuration Changes

- System configuration:
  - <server-id, address> for each server
  - Determines what constitutes a majority
- Consensus mechanism must support changes in the configuration:
  - Replace failed machine
  - Change degree of replication

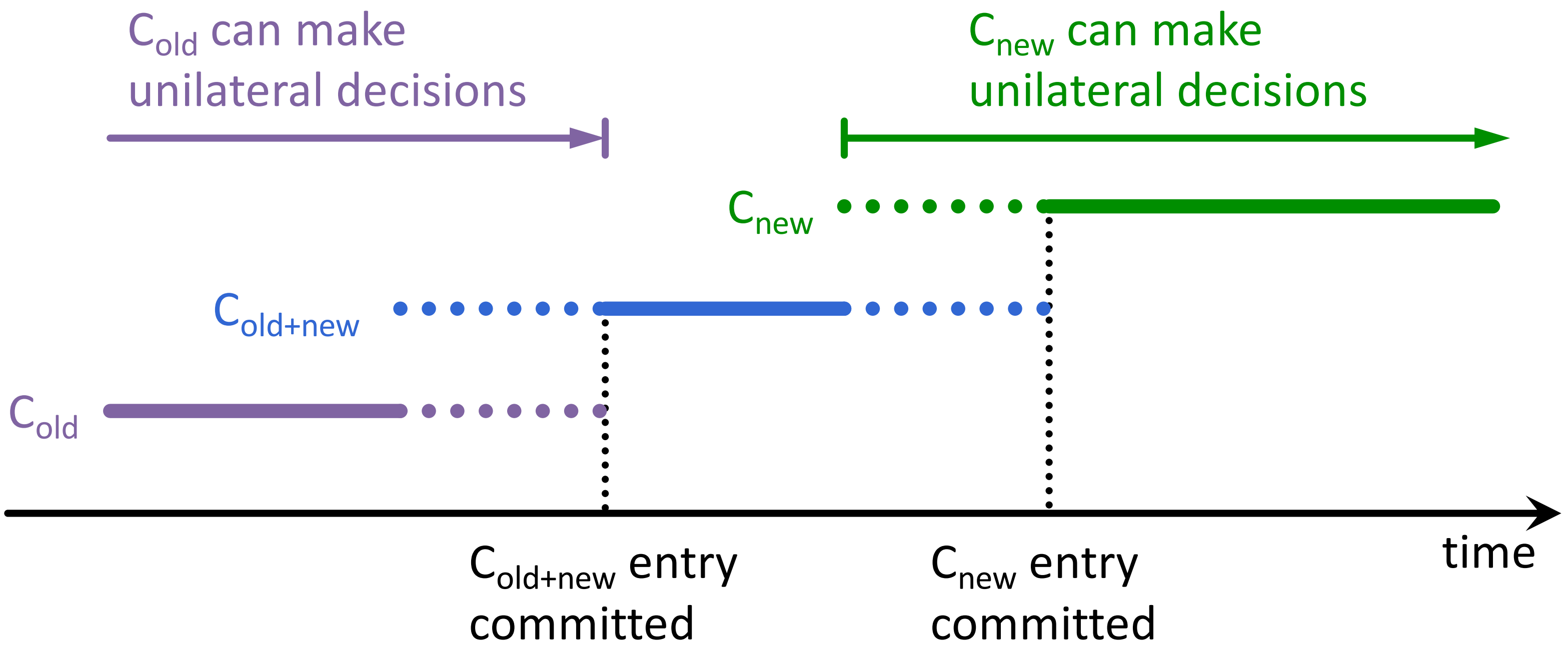
Configuration Changes, cont'd

Cannot switch directly from one configuration to another: conflicting majorities could arise



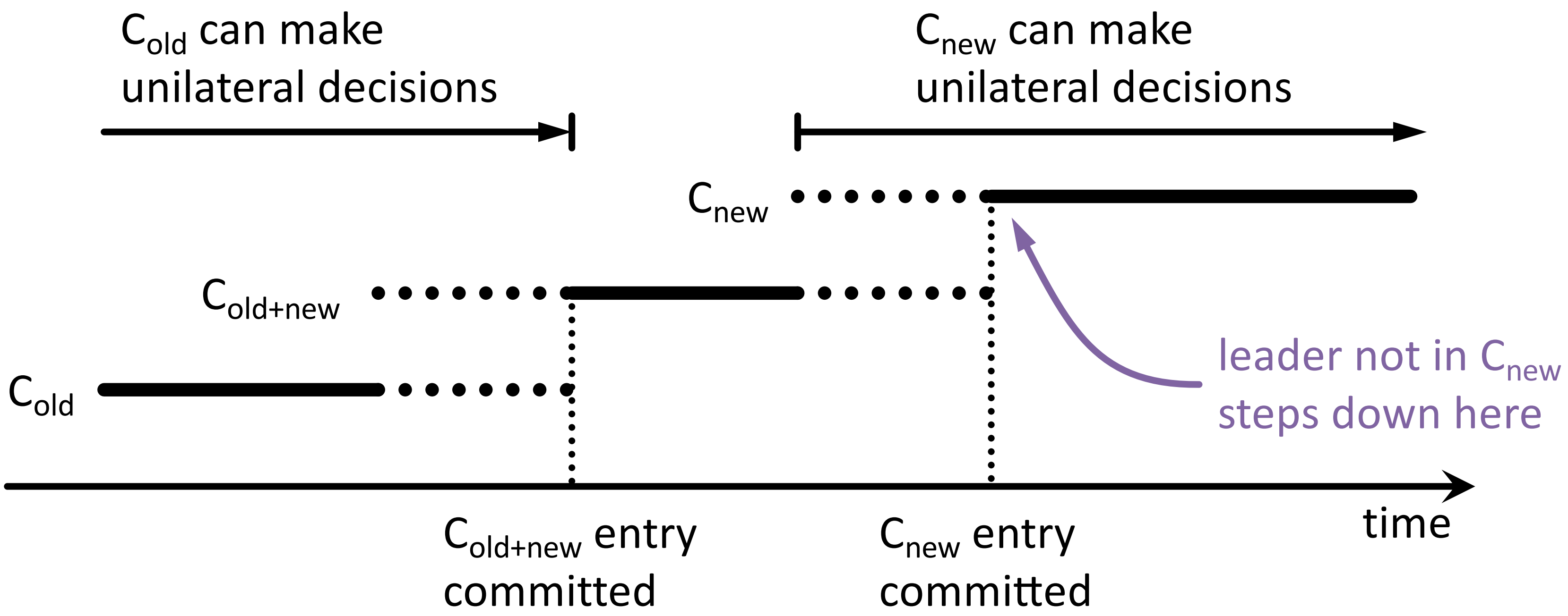
Joint Consensus

- Raft uses a 2-phase approach:
  - Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
  - Configuration change is just a log entry; applied immediately on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration



Joint Consensus, cont’d

- Any server from either configuration can serve as leader
- If current leader is not in C_{new}, must step down once C_{new} is committed.



C_{old} can make
unilateral decisions

C_{new} can make
unilateral decisions

C_{new}

$C_{old+new}$

C_{old}

leader not in C_{new}
steps down here

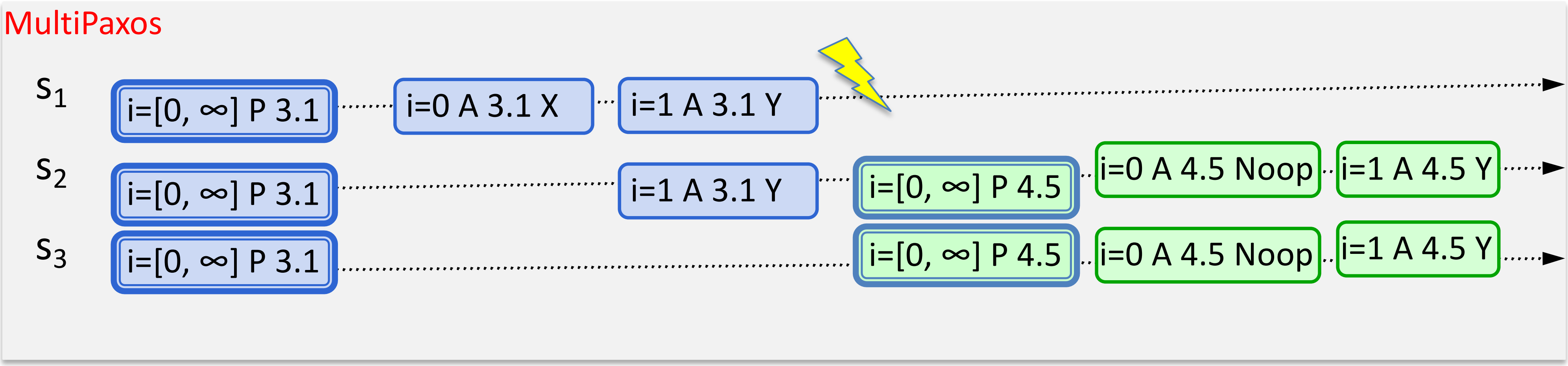
$C_{old+new}$ entry
committed

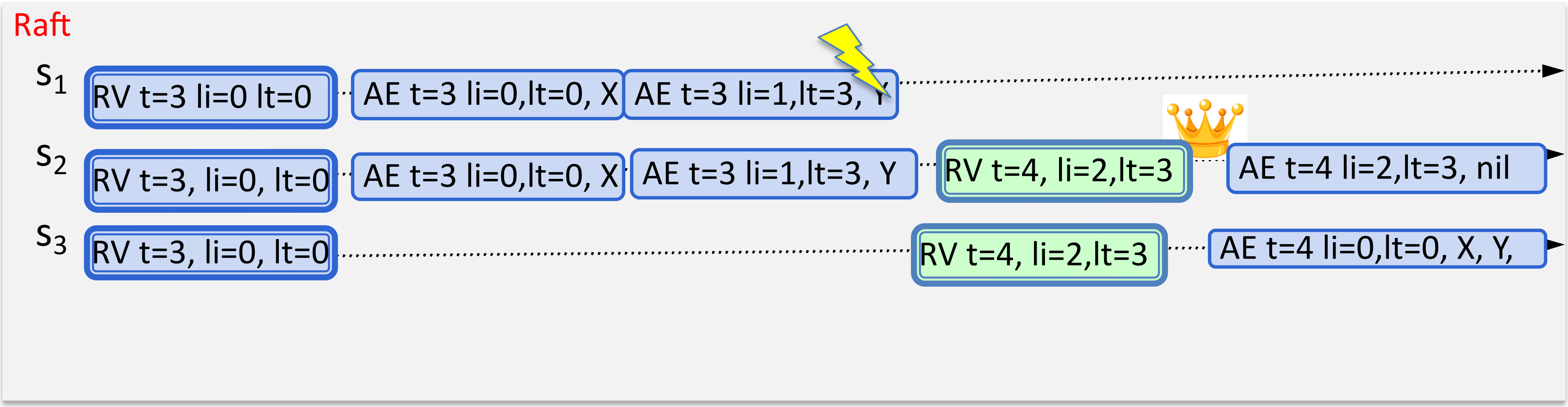
C_{new} entry
committed

time

Paxos vs. Raft

- Different protocols? or variants of the same thing?





(Multi-)Paxos

Each Server's State:

Proposer identifier

Ballot number

Monotonically increasing

Ballot number is another name for proposal number <local_counter: server-id>

Instances

value$_1$	value$_2$	value$_3$
ballot$_1$	ballot$_2$	ballot$_3$

(Multi-)Paxos

Each Server's State:

Proposer identifier

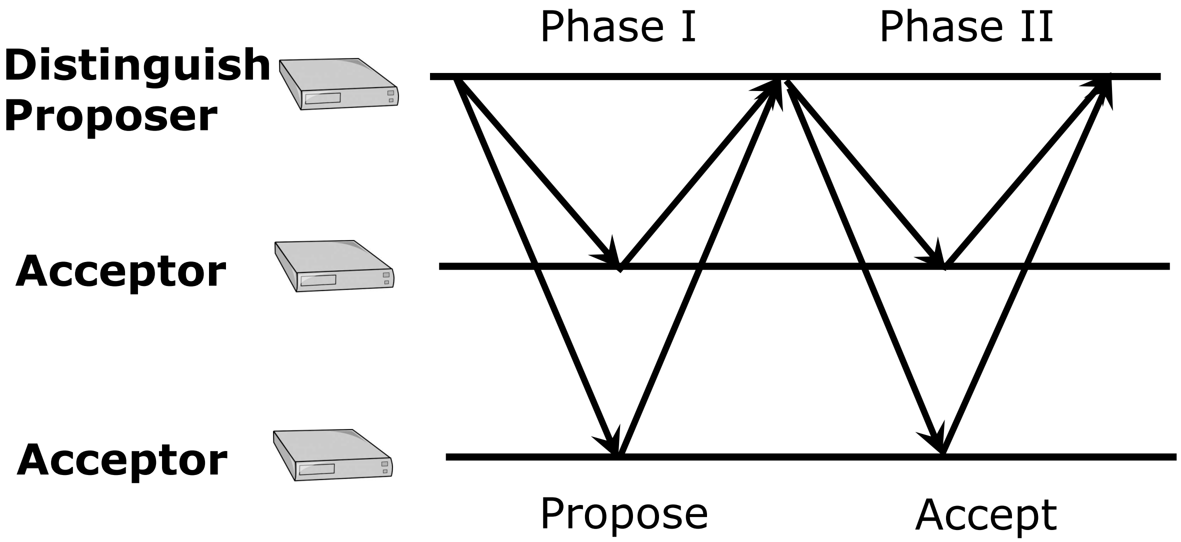
Ballot number

Monotonically increasing

Instances

value$_1$	value$_2$	value$_3$
ballot 1	ballot 2	ballot 3

Workflow



Phase I

Phase II

Distinguish Proposer

Acceptor

Acceptor

Propose

Accept

Raft

Each Server's State:

Leader identifier

Term number

Monotonically increasing

Log

value$_1$	value$_2$	value$_3$
term$_1$	term$_2$	term$_3$

Raft

Each Server's State:

Leader identifier

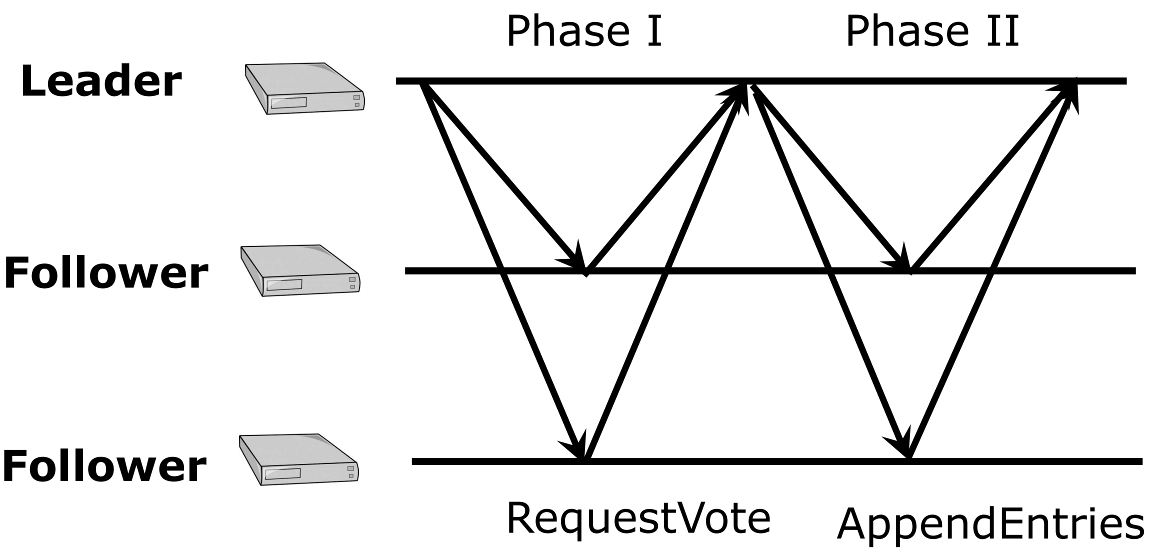
Term number

Monotonically increasing

Log

value$_1$	value$_2$	value$_3$
term$_1$	term$_2$	term$_3$

Workflow



Phase I

Phase II

Leader

Follower

Follower

RequestVote

AppendEntries

Intuition – They Are Similar (equivalent?)

Similar State

Ballot ⇔ Term

Instance ⇔ Log Entry

Instance.ballot ⇔ Entry.term

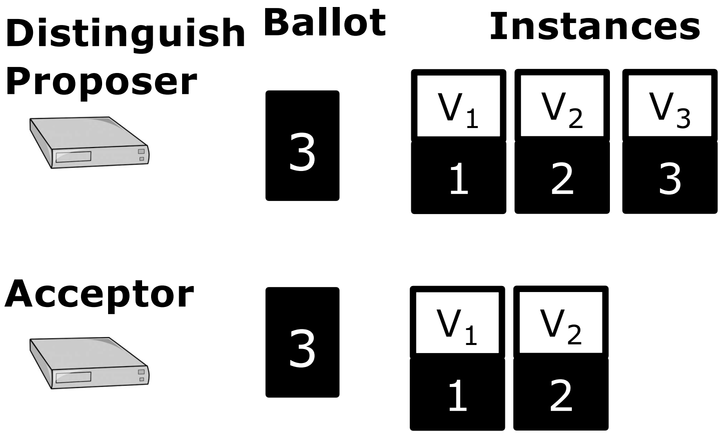
Similar Workflow

Propose ⇔ RequestVote

Accept ⇔ AppendEntries

They are similar, but not equivalent
(in terms of their state transitions)

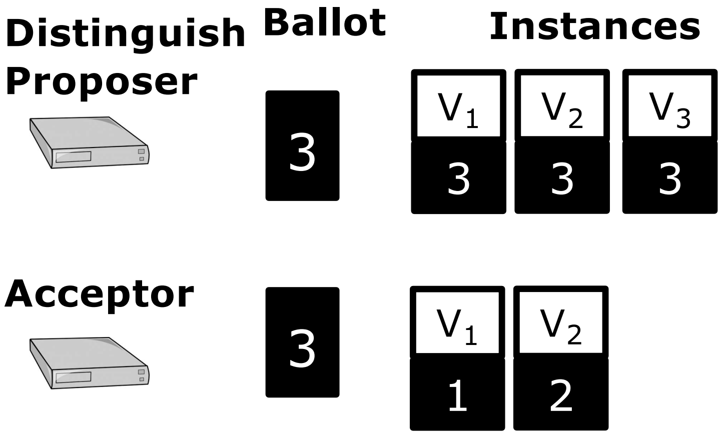
The Difference – Example I



	Ballot	Instances
Distinguish Proposer	3	V_1 1, V_2 2, V_3 3
Acceptor	3	V_1 1, V_2 2

Paxos

The Difference – Example I



Distinguish Proposer

Ballot

Instances

3

V_1 3 | V_2 3 | V_3 3

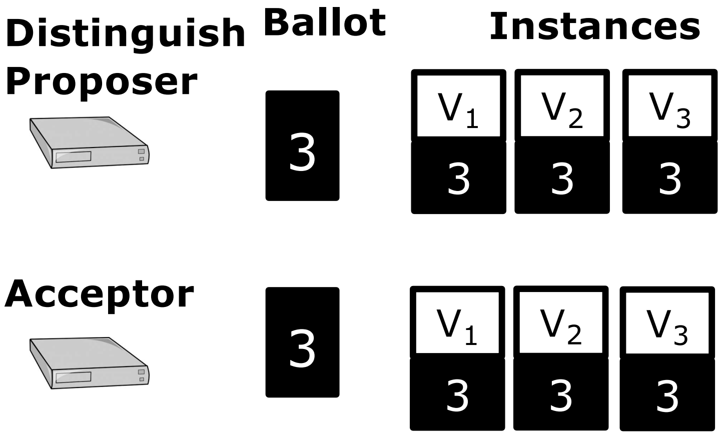
Acceptor

3

V_1 1 | V_2 2

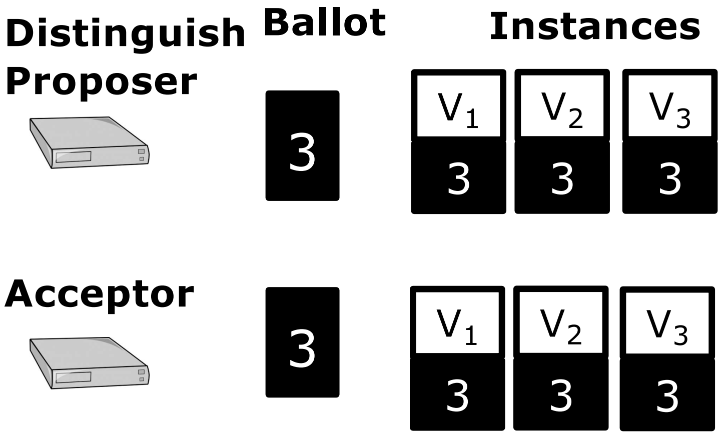
Paxos

The Difference – Example I

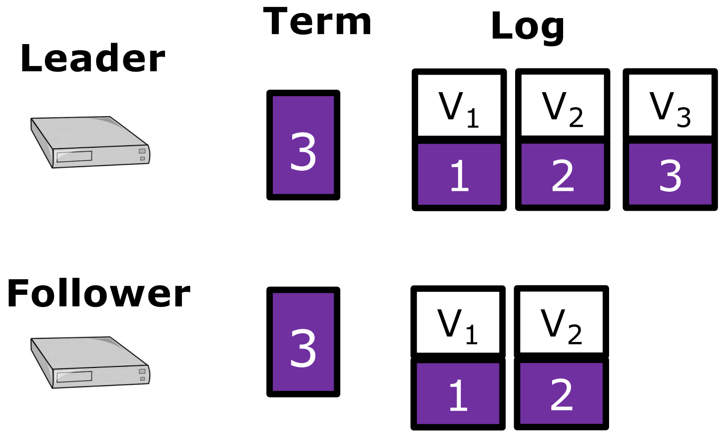


Paxos

The Difference – Example I

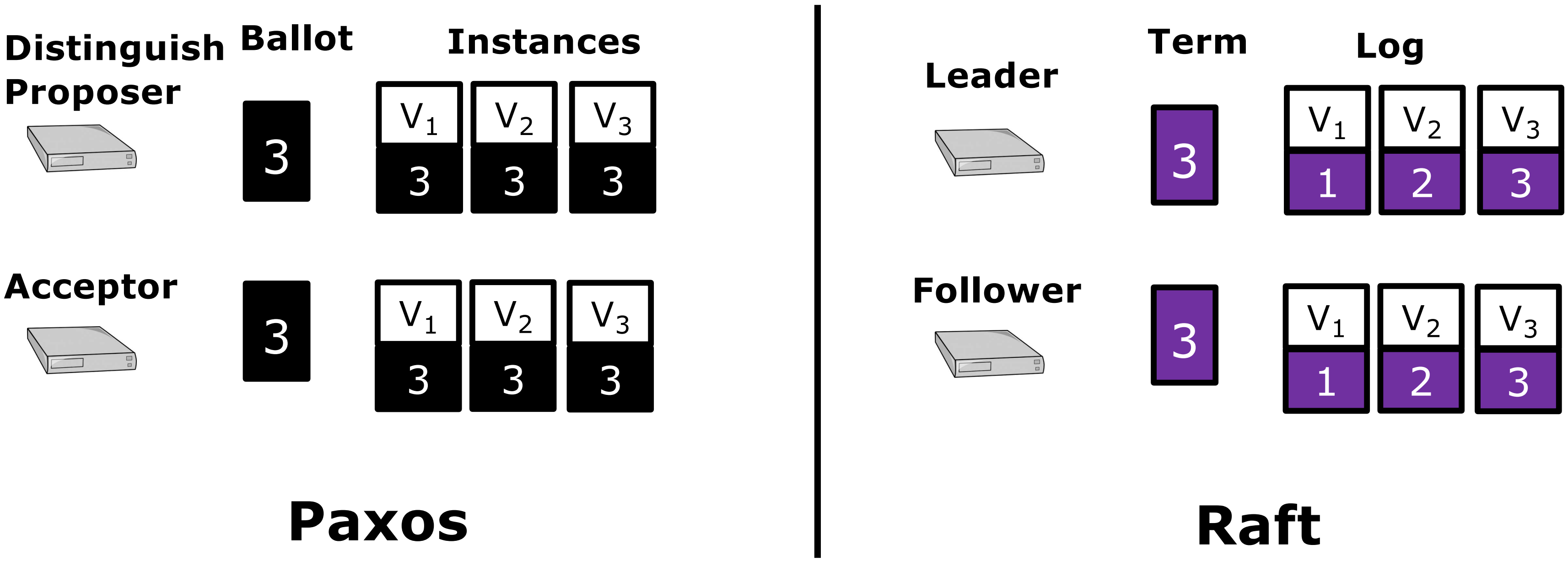


Paxos



Raft

The Difference – Example I



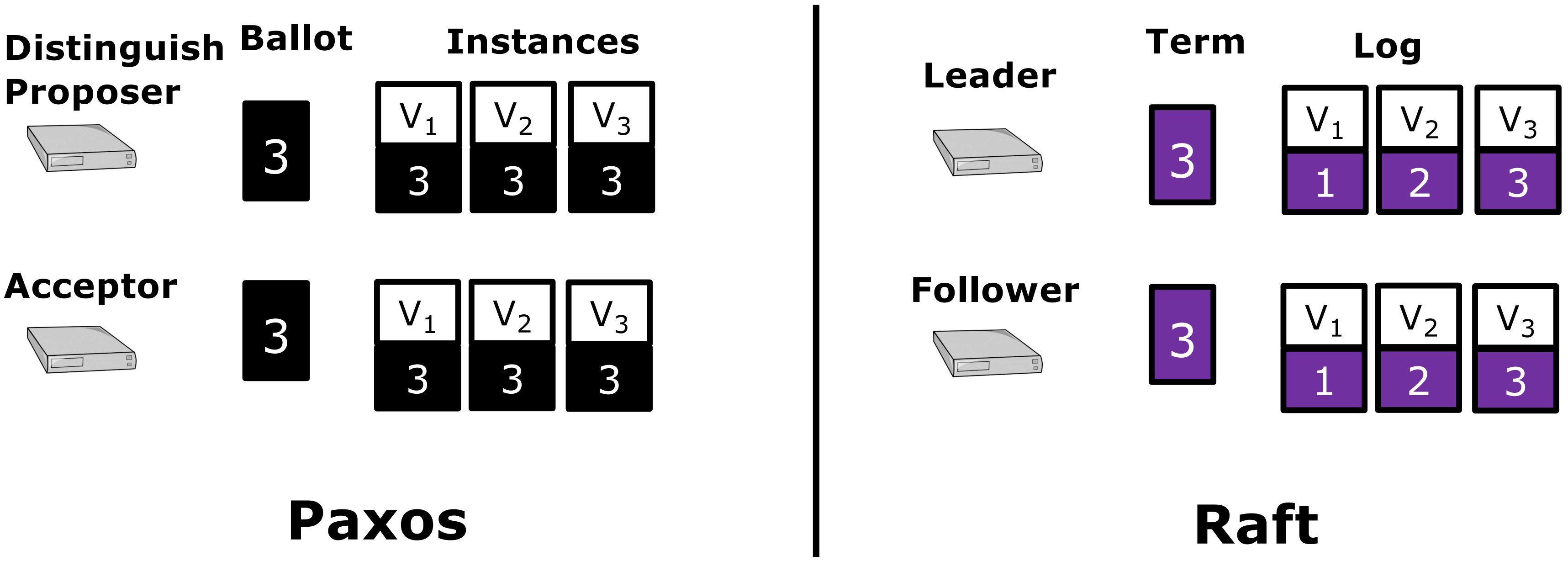
	Ballot	Instances		
Distinguish Proposer	3	V_1 3	V_2 3	V_3 3
Acceptor	3	V_1 3	V_2 3	V_3 3

Paxos

	Term	Log		
Leader	3	V_1 1	V_2 2	V_3 3
Follower	3	V_1 1	V_2 2	V_3 3

Raft

The Difference – Example I



	Ballot	Instances		
		V_1	V_2	V_3
Distinguish Proposer	3	3	3	3
Acceptor	3	3	3	3

Paxos

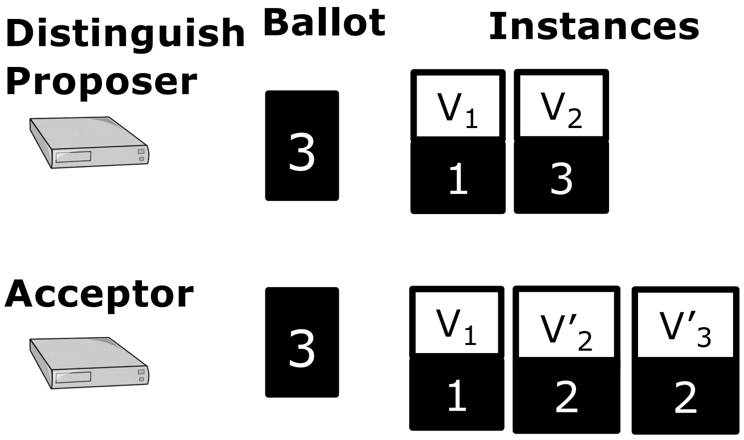
	Term	Log		
		V_1	V_2	V_3
Leader	3	1	2	3
Follower	3	1	2	3

Raft

Ballot of uncommitted instances will be updated in accept phase

Term of uncommitted entries never change in append phase

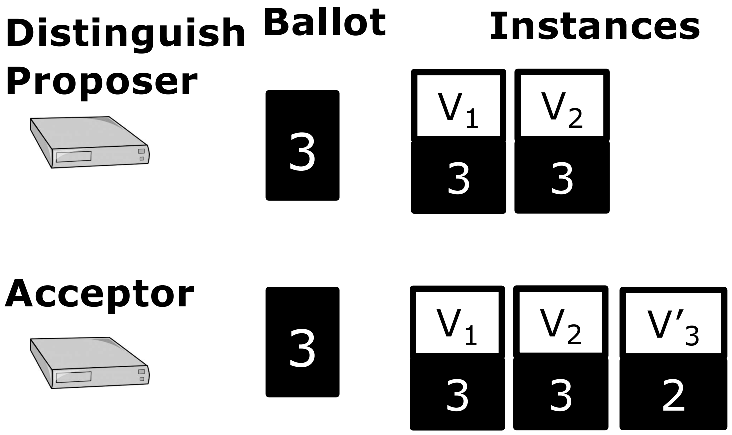
The Difference – Example II



	Ballot	Instances		
Distinguish Proposer	3	V_1 1	V_2 3	
Acceptor	3	V_1 1	V'_2 2	V'_3 2

Paxos

The Difference – Example II



Distinguish Proposer

Ballot

Instances

3

V_1 3

V_2 3

Acceptor

3

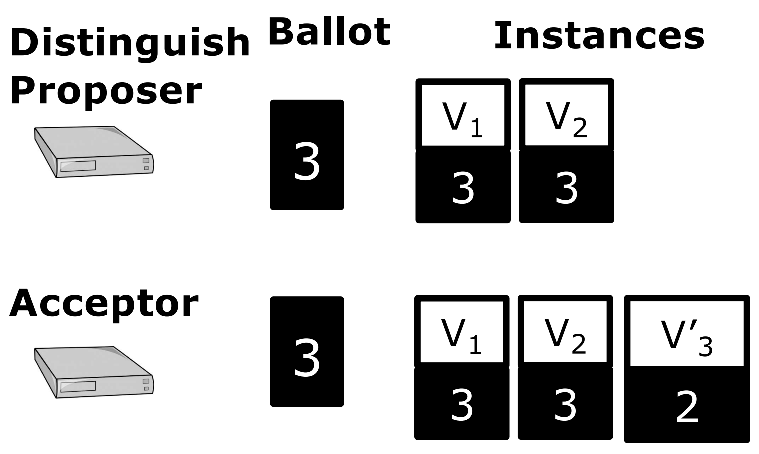
V_1 3

V_2 3

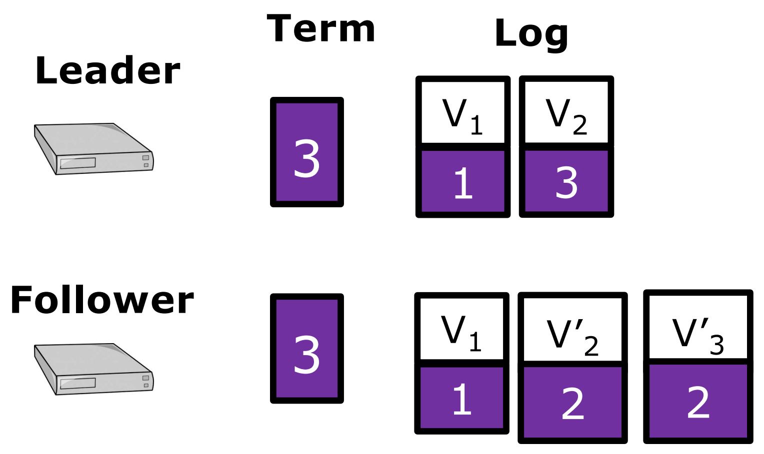
V'_3 2

Paxos

The Difference – Example II

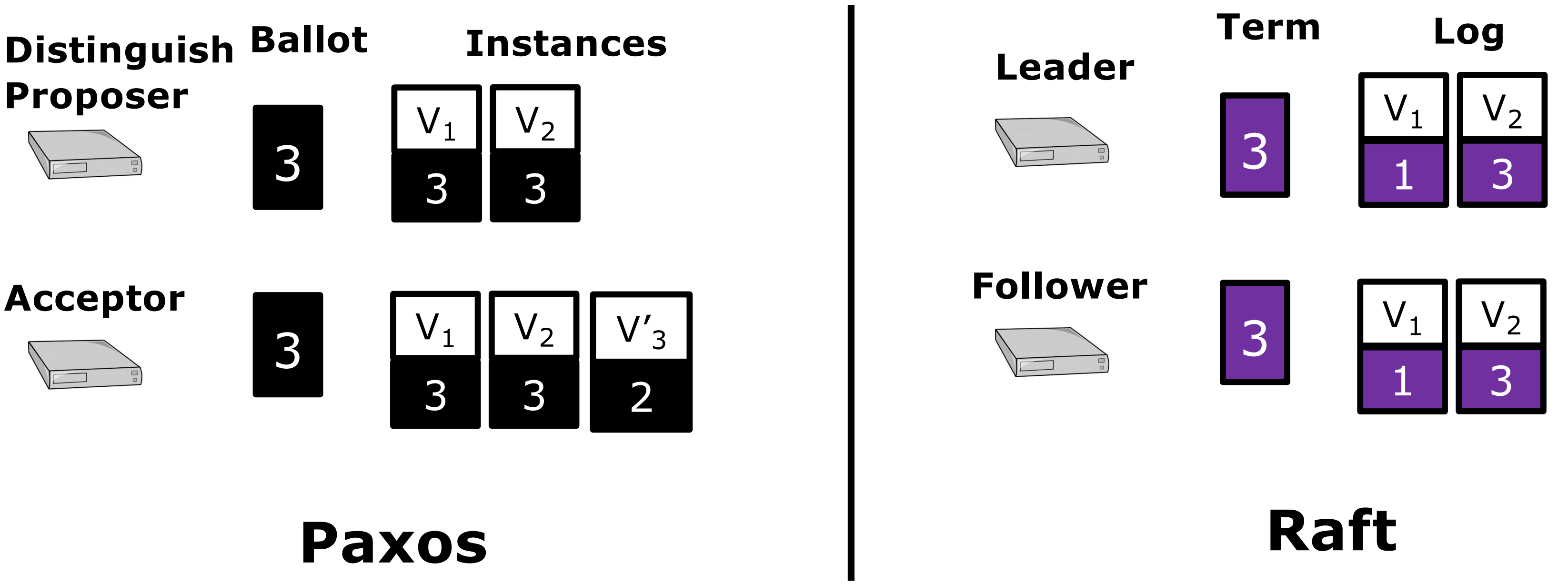


Paxos



Raft

The Difference – Example II



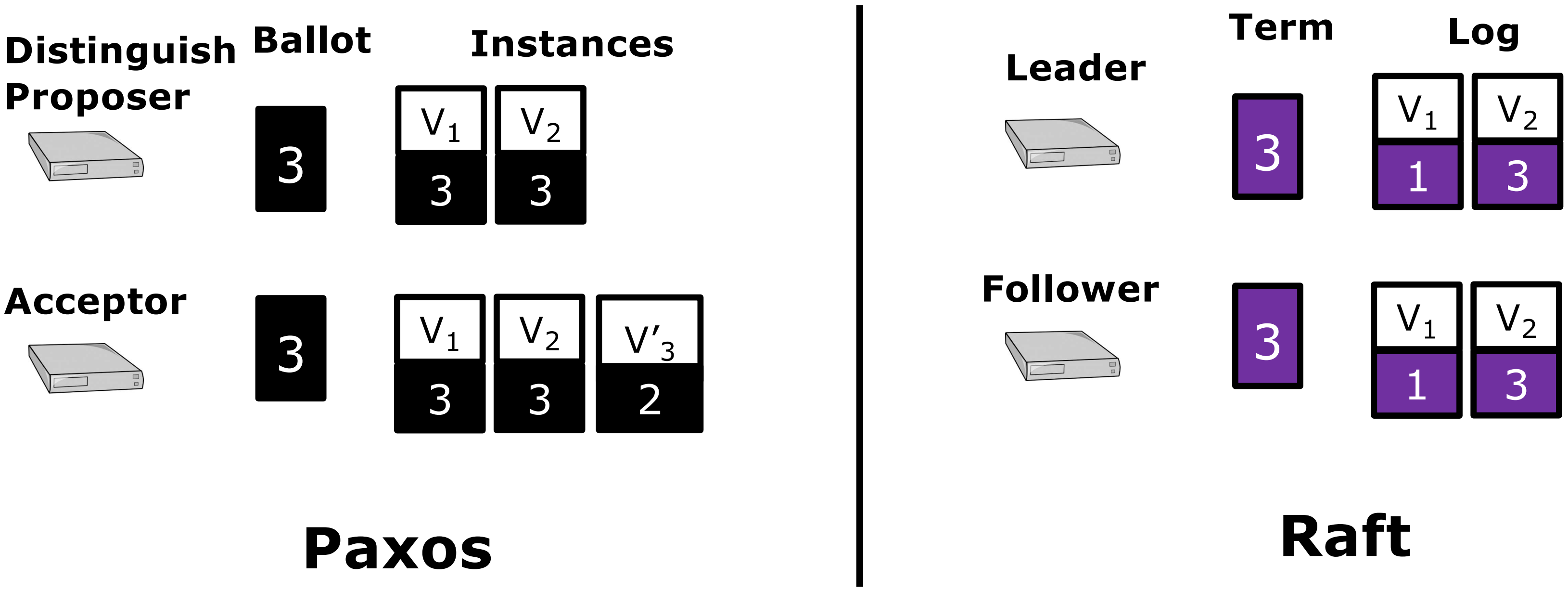
	Ballot	**Instances**		
Distinguish Proposer	3	V_1 3	V_2 3	
Acceptor	3	V_1 3	V_2 3	V'_3 2

Paxos

	Term	**Log**	
Leader	3	V_1 1	V_2 3
Follower	3	V_1 1	V_2 3

Raft

The Difference – Example II



Paxos does not delete accepted instance

Raft deletes appended entries on demand.

The Differences

Accepted Instance	≠	**Appended Log Entry**
Accept ballot can be updated	↔	Append term is read only
Acceptor cannot delete accepted instances	↔	Follower can delete its log entries

Paxos commit point:
Proposal has been accepted
by a majority

Raft commit point:
Entry has been appended
at majority && a later entry with
currentTerm has been appended
at majority