CE 528 Cloud Computing

Lecture 6: Consensus
Spring 2026

Prof. Yigong Hu

Slides courtesy of Chang Lou and Robert Morrios

Administrivia

Share your AWS ID by Tonight

- https://forms.gle/C9vuYuMkDJcN2xEF7

Motivation

A pattern in the fault-tolerant systems:

- MR replicates computation but relies on a single master to organize
- GFS replicates data but relies on the master to pick primaries

A single entity to make decision

- Avoid split brain
- But the single machine can have failure

Recap: Split Brain

In GFS, if the primary crashes:

- The coordinator must be able to designate a new primary if the present primary fails.
- But the coordinator cannot distinguish "primary has failed" from "primary is still alive but the network has a problem."
- What if the coordinator designates a new primary while old one is active?
  - two active primaries!
  - C1 writes to P1, C2 reads from P2, doesn't seen C1's write!

“Server Crashed” vs “Network Broken”

The symptom is the same: no response to a query over the network

The bad situation is often called "network partition":

- C1 can talk to S1, C2 can talk to S2,
- but C1+S1 see no responses from C2+S2

Solution?

- a perfectly reliable network
- human to decide

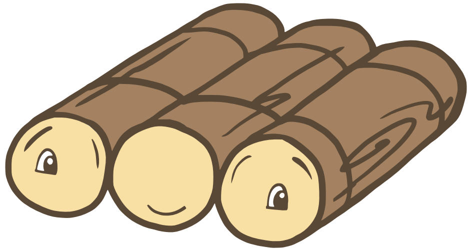
Consensus Protocols

Require only a majority of nodes to be up at any time in order to make progress.

Two papers:

- Paxos: [Lamport-1998]: Original protocol. Solves the basic consensus problem
- Raft: More recent, operates at a higher level of abstraction

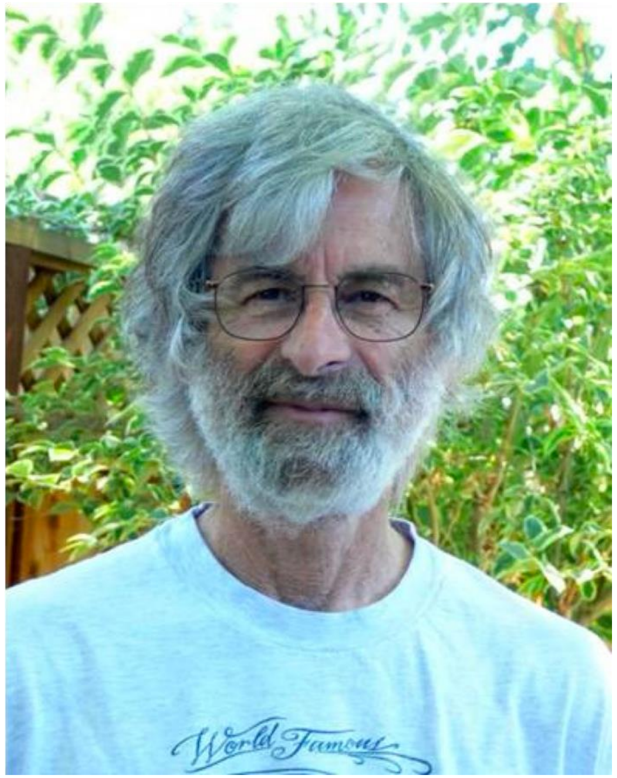
Paxos



Leslie Lamport

Winner of the 2013 Turing Award

Known for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems



Also known for

- LaTeX (typesetting system)
- Byzantine Fault Tolerance
- TLA+ (formal verification tools)
- …

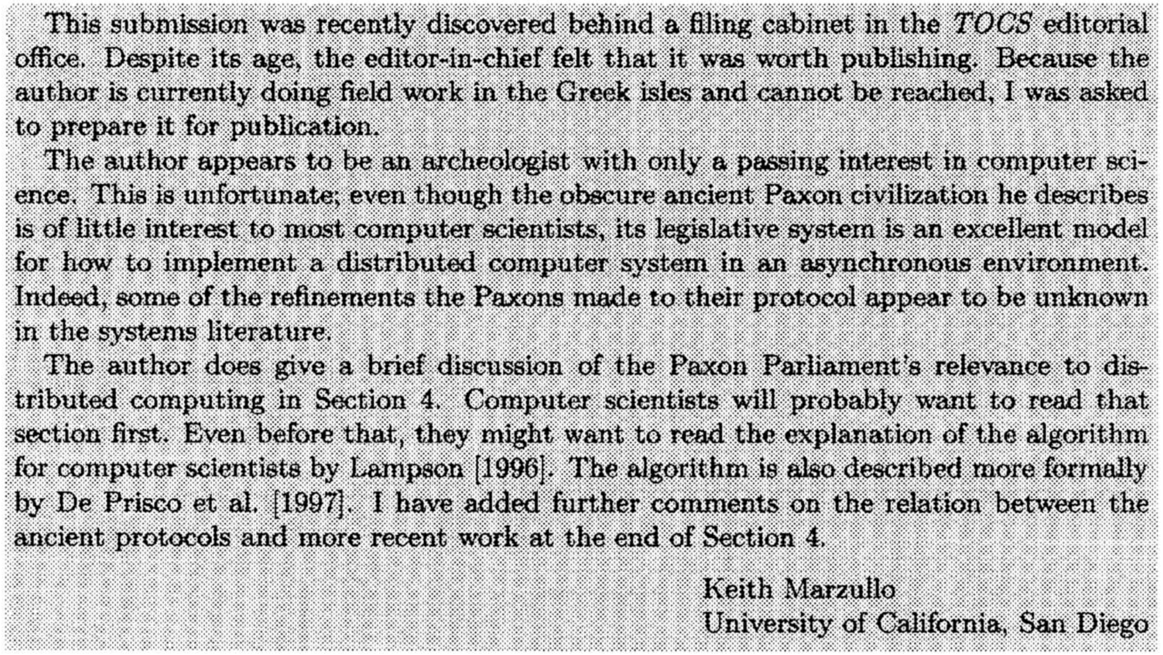
A Brief History of Paxos

1989	Viewstamped Replication – Liskov & Oki
1990	Paxos – Leslie Lamport, “**The Part-Time Parliament**”
1998	Paxos paper published
~2005	First practical deployments
2010s	Widespread use!
2014	Lamport wins Turing Award

The Part-time Parliament

First submitted in 1990, published in 1998, won ACM SIGOPS Hall of Fame Award in 2012

The initial draft was obscure with an "archaeological" tone

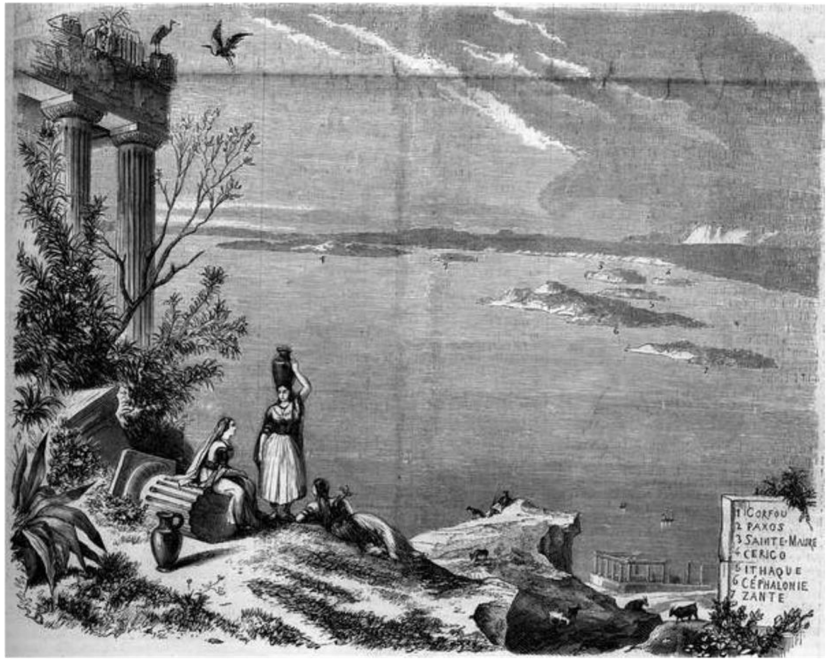


This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

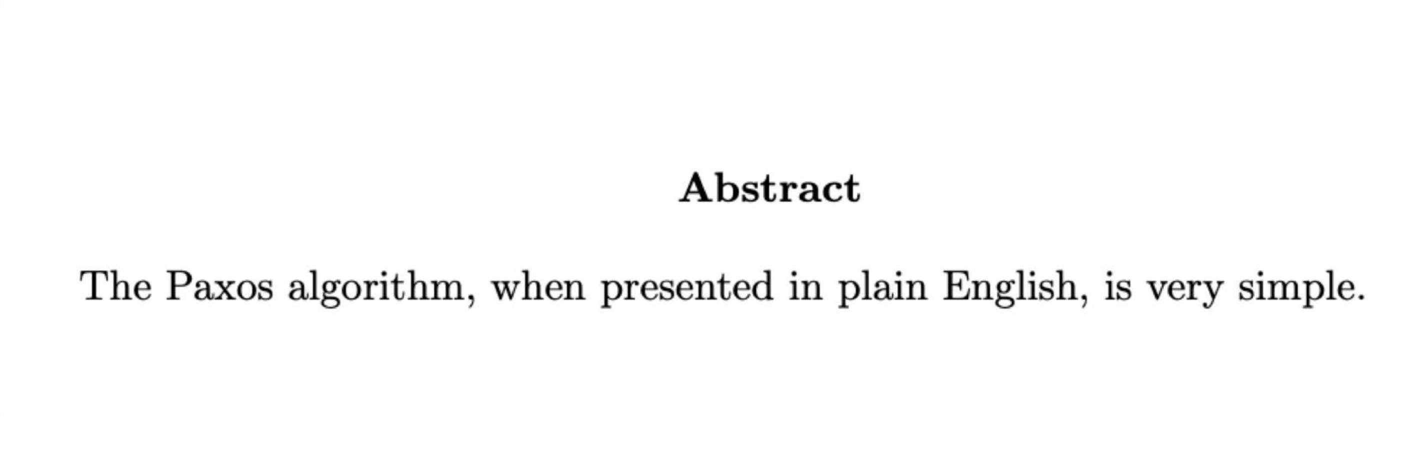
Keith Marzullo
University of California, San Diego



Paxos Made Simple

Three years after publishing original Paxos paper, Lamport published a simplified version to explain the protocol

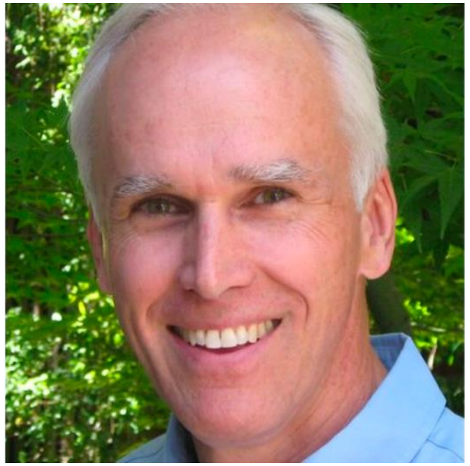
- "The current version is 13 pages long, and contains no formula more complicated than n1 > n2."
- Maybe a bit over-simplified..



Abstract

The Paxos algorithm, when presented in plain English, is very simple.

Raft



John Ousterhout
Professor of Computer Science
Stanford University

Diego Ongaro
Ph.D., Computer Science
Stanford University

Majority Vote

Agreement from a majority is required to do anything

- f+1 out of 2f+1

Why does majority help avoid split brain?

- at most one partition can have a majority

A key property of majorities is that any two must intersect

- e.g. successive majorities for Raft leader election must overlap
- and the intersection can convey information about previous decisions

A Preview of Raft with Visualization

https://thesecretlivesofdata.com/raft/

Server States

At any given time, each server is either:

- Leader: handles all client interactions, log replication
- Follower: completely passive
- Candidate: used to elect a new leader

Normal operation: 1 leader, N-1 followers

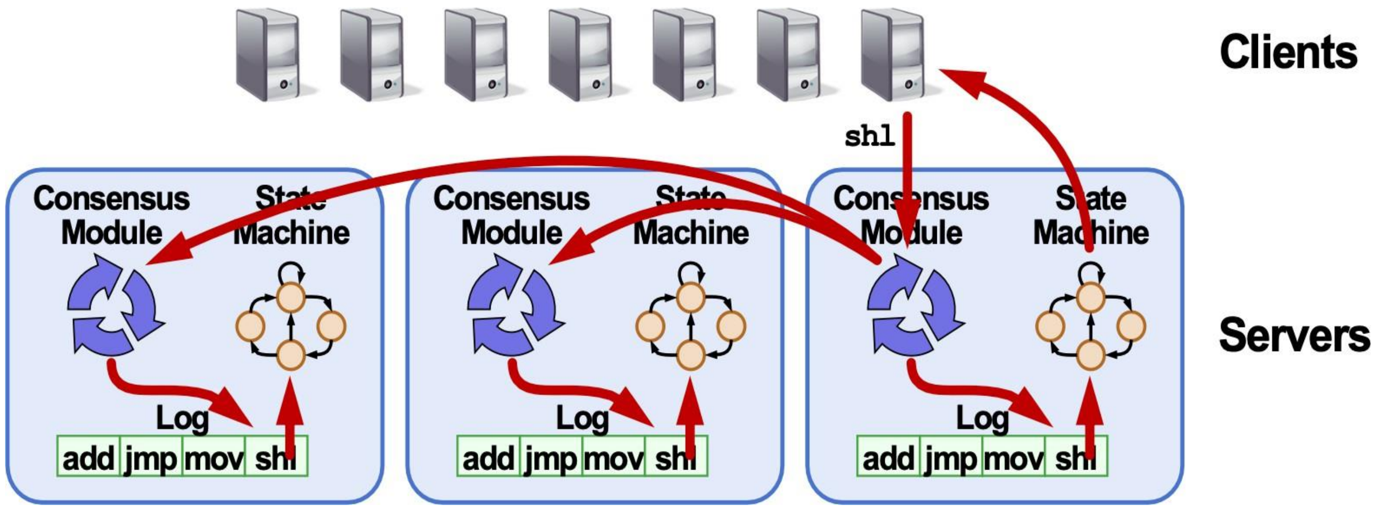
- Exercise: how to states transit to others

Follower

Candidate

Leader

Raft Overview

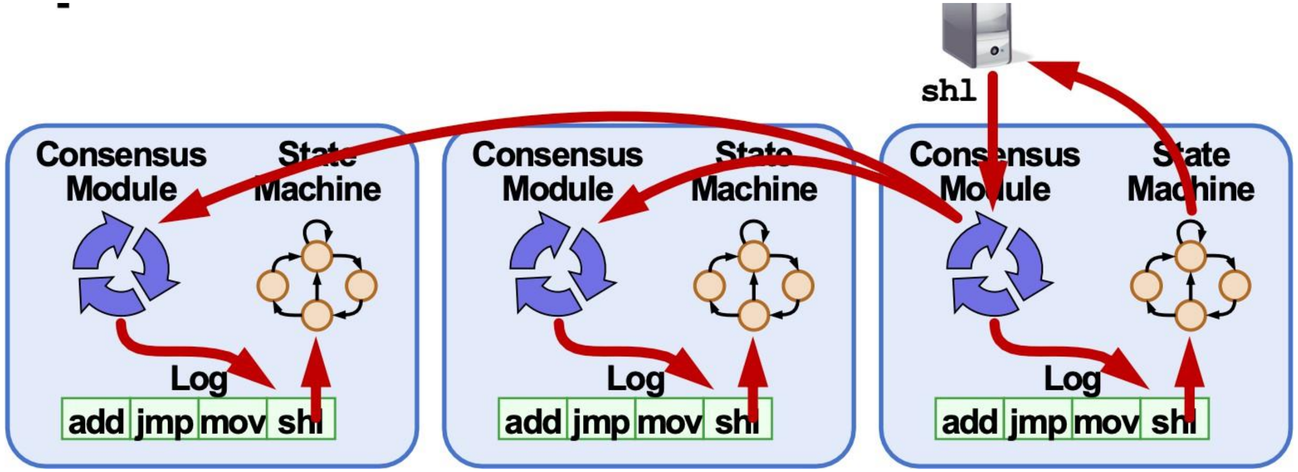


Replicated log => replicated state machine

- All servers execute same commands in same order

Consensus module ensures proper log replication

Normal Operation



Client sends command to leader

Leader appends command to its log

Leader sends AppendEntries RPCs to followers

Once new entry committed:

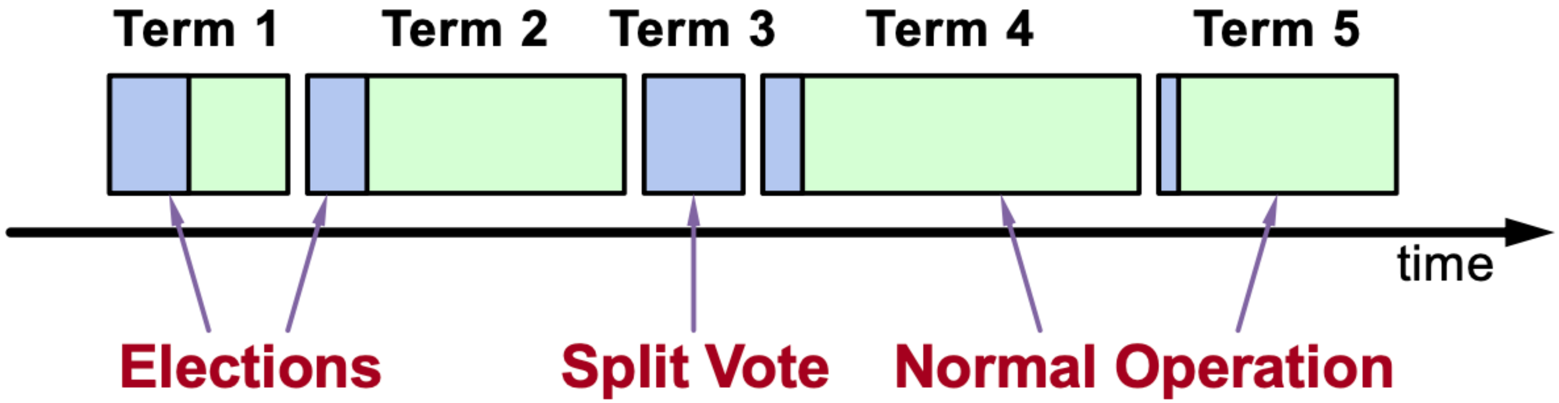
- Leader passes command to its state machine, sends result to client
- Leader piggybacks commitment to followers in later AppendEntries
- Followers pass committed commands to their state machines

Leader

Why a leader?

- ensures all replicas execute the same commands, in the same order

Term (aka Epochs)



Raft divides time into terms:

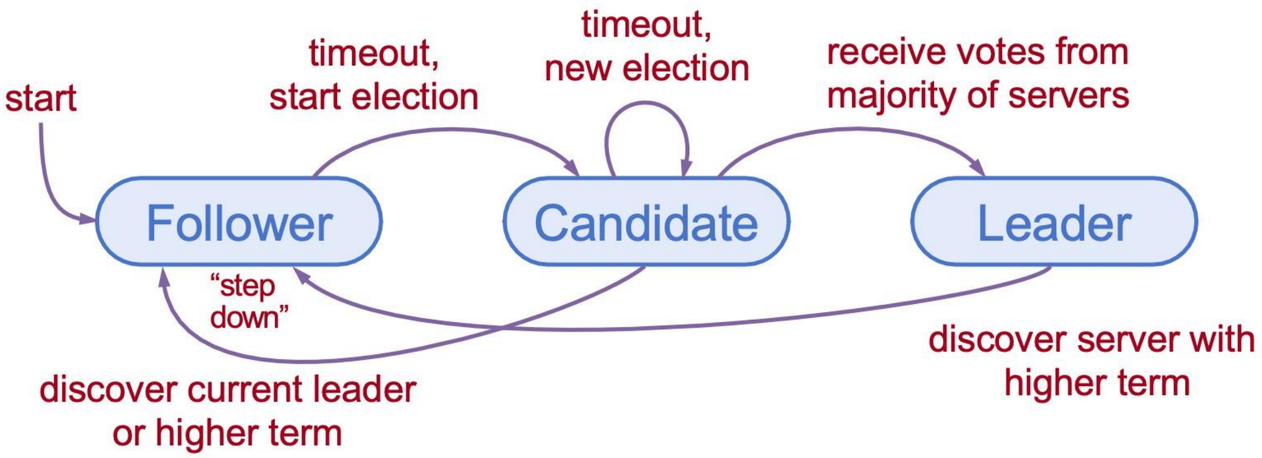
- In each term, election (new leader -> new term)
- A term has at most one leader; might have no leader

Each server maintains current term value

Key role of terms: identify obsolete information

Leader State Transition

1. Servers start as followers
2. Leaders send heartbeats (empty AppendEntries RPCs) to maintain authority over followers
3. If election Timeout elapses with no RPCs (100-500ms), follower assumes leader has crashed and starts new election



Elections

Start election:

- Increment current term, change to candidate state, vote for self

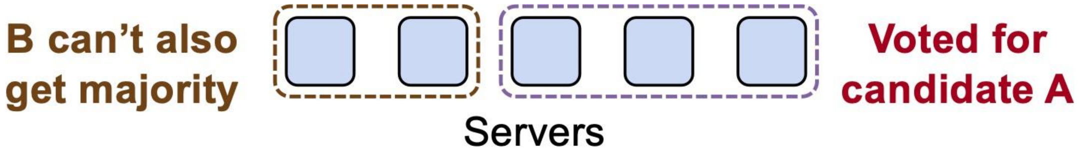
Send Request Vote to all other servers, retry until either:

1. Receive votes from majority of servers:
    - Become leader
    - Send AppendEntries heartbeats to all other servers
2. Receive RPC from valid leader:
    - Return to follower state
3. No-one win selection (election timeout elapses):
    - Increment term, start new election

Elections

Safety: allow at most one winner per term

- Each server votes only once per term (persists on disk)
- Two different candidates can’t get majorities in same term



Liveness: some candidate eventually wins

- Each choose election timeouts randomly in [T, 2T]
- One usually initiates and wins election before others start
- Works well if T >> network RTT

Election Success

How does a server learn about newly elected leader?

- new leader sees yes votes from majority
- others see AppendEntries heart-beats with a higher term number
    - i.e. from the new leader
- the heart-beats suppress any new election

An election may not succeed for two reasons:

- less than a majority of servers are reachable
- simultaneous candidates split the vote, none gets majority

Solution: Randomization

- each server picks a random election timeout
- how to choose the election timeout?
    - at least a few heartbeat intervals
    - long enough to let one candidate succeed

Why Log?

The log orders the commands

- to help replicas agree on a single execution order
- to help the leader ensure followers have identical logs

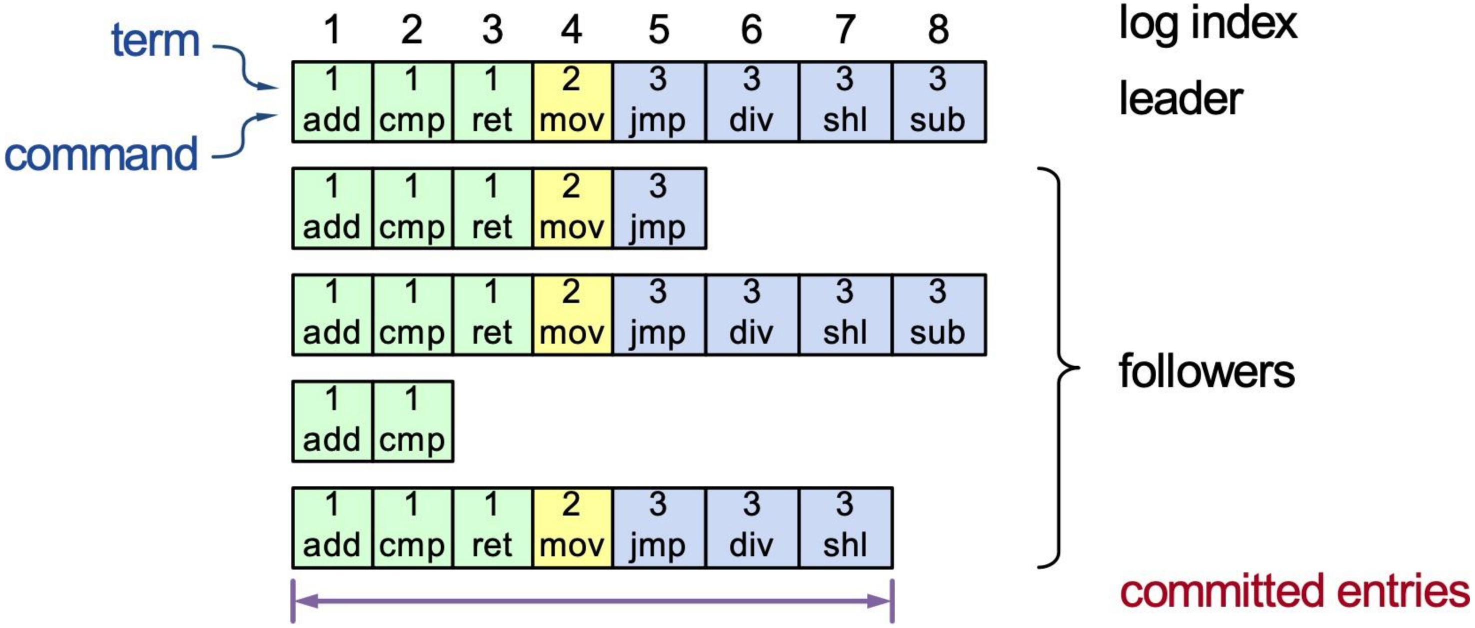
The log stores tentative commands until committed

The log stores commands in case leader must re-send to followers

The log stores commands persistently for replay after reboot

Are the servers' logs exact replicas of each other?

Log Structure



term

command

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

committed entries

Log entry = < index, term, command >

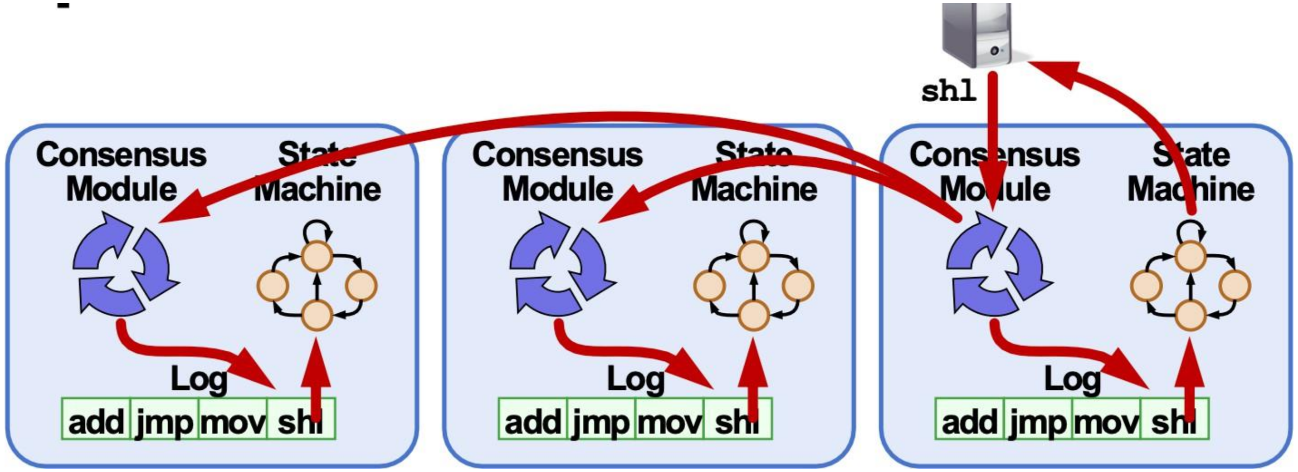
Log stored on stable storage (disk); survives crashes

Entry committed -> stored on majority of servers

- Durable / stable, will eventually be executed by state machines

Crash

-

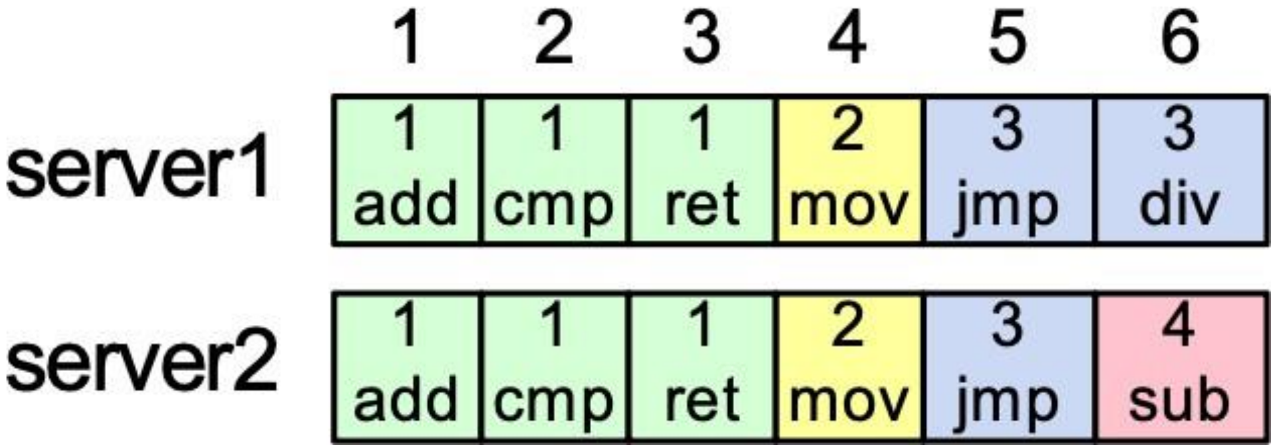


Crashed / slow followers?

Performance is “optimal” in common case:

- Leader retries RPCs until they succeed
- Followers pass committed commands to their state machines

Log Operation



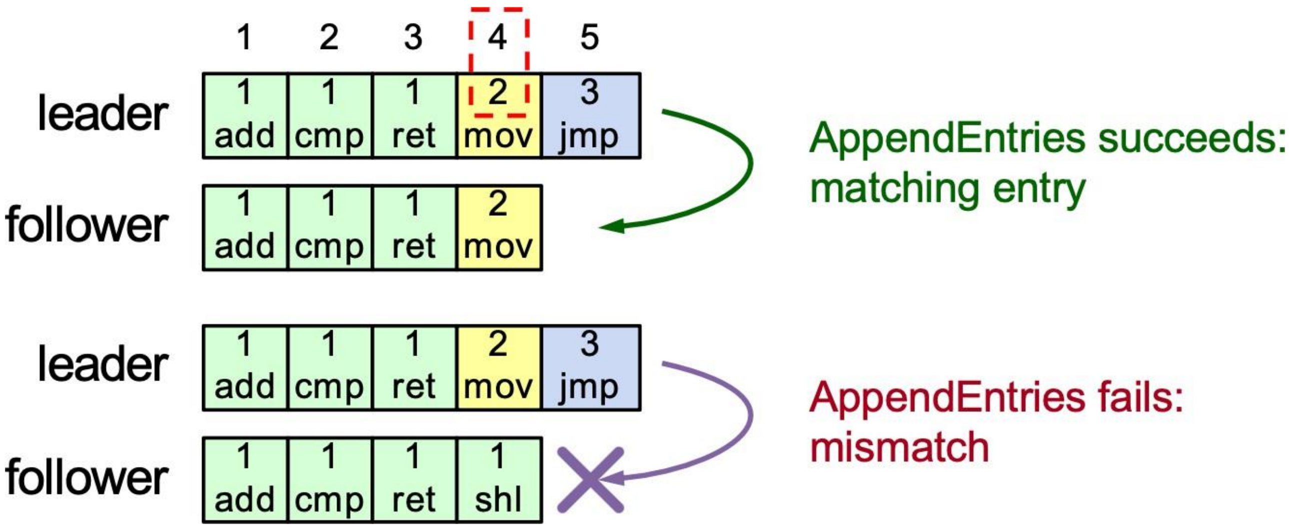
	1	2	3	4	5	6
server1	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
server2	1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

If log entries on different server have same index and term:

- Store the same command => one leader, one entry, one index, one term + log position never change
- Logs are identical in all preceding entries => consistency check

If given entry is committed, all preceding also committed

Consistency Check



	1	2	3	4	5
leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	2 mov	

AppendEntries succeeds:
matching entry

leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	1 shl	

AppendEntries fails:
mismatch

AppendEntries has of entry preceding new ones

Follower must contain matching entry; otherwise it rejects

Implements an induction step, ensures coherency

Leader Changes

As long as the leader stays up:

- clients only interact with the leader
- clients can't see follower states or logs

Things get interesting when changing leaders

- e.g. after the old leader fails
- how to change leaders without anomalies?

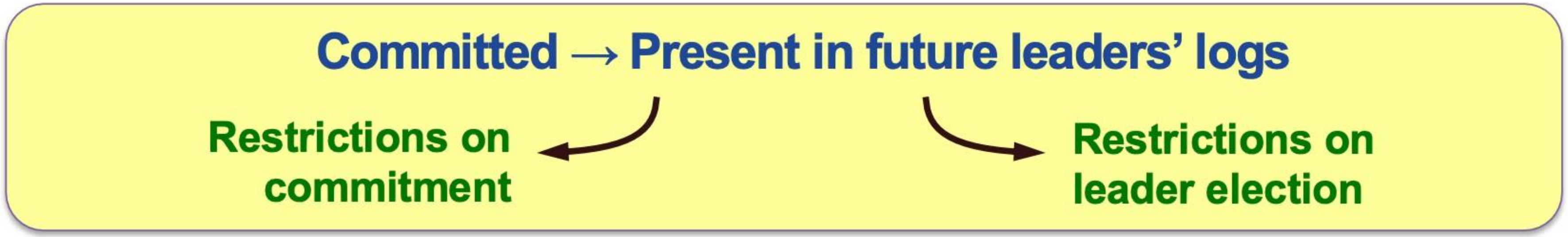
Safety Requirement

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

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

Why does this guarantee higher-level goal?

1. Leaders never overwrite entries in their logs
2. Only entries in leader’s log can be committed
3. Entries must be committed before applying to state machine



Committed → Present in future leaders’ logs

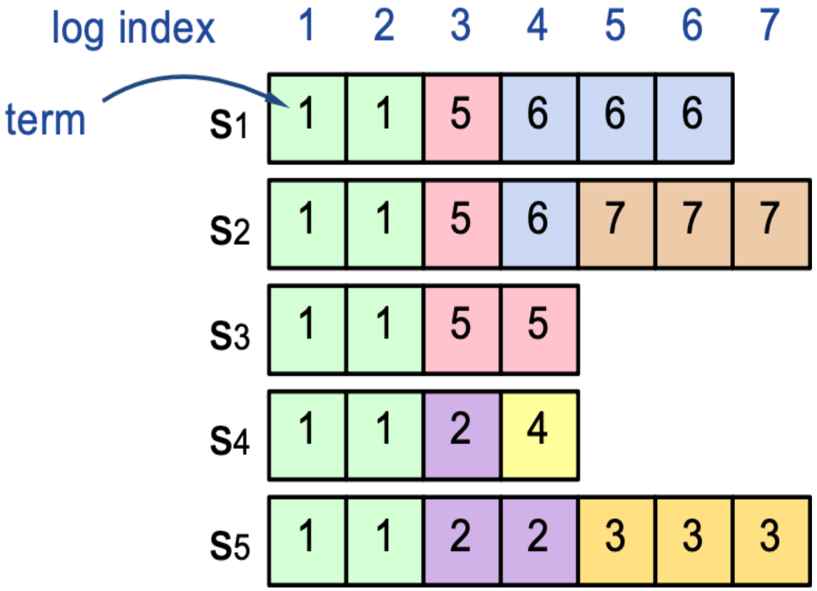
Restrictions on commitment

Restrictions on leader election

Crashes can Leave Log Disagree

- **A leader crashes before sending last AppendEntries to all**

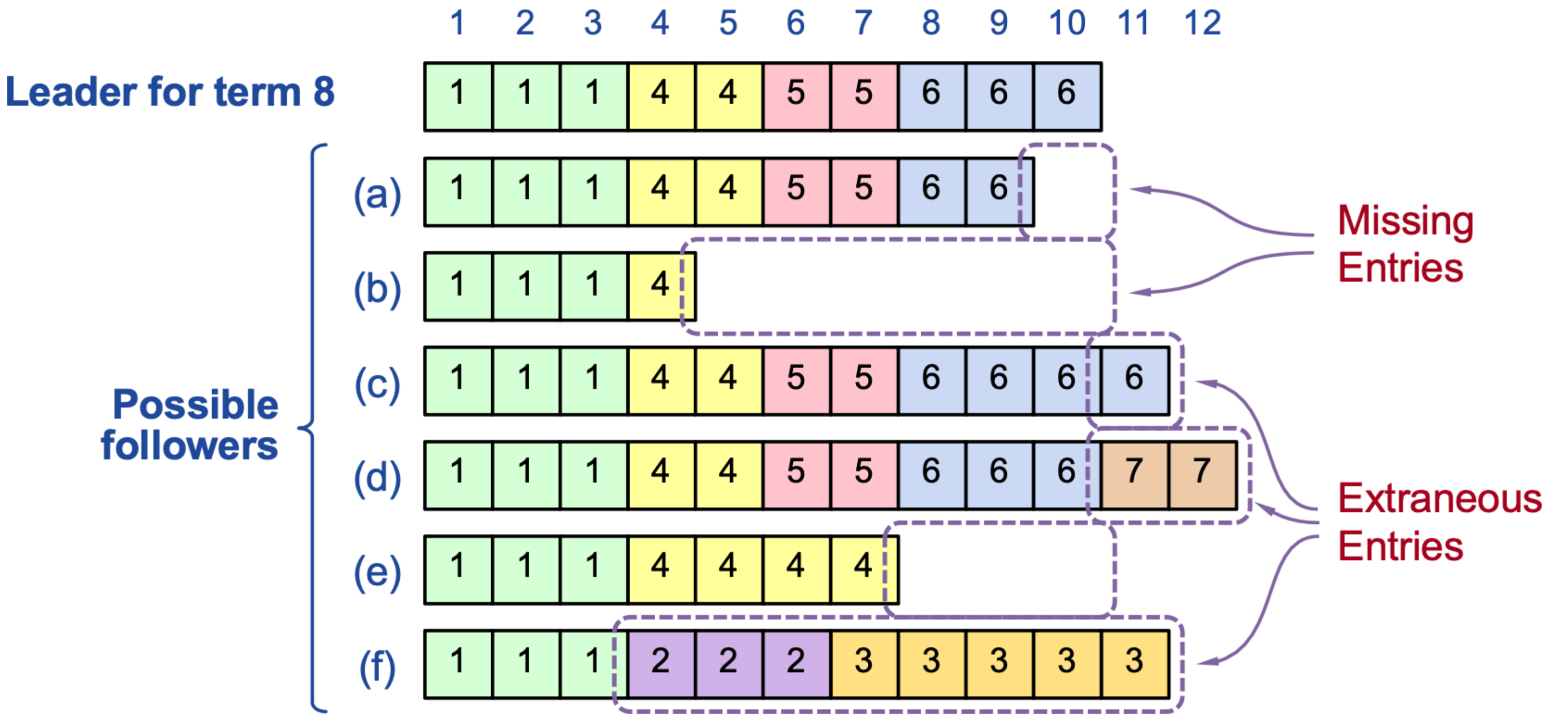
- **Logs might have different commands in same entry! after a series of leader crashes**



term

log index	1	2	3	4	5	6	7
S1	1	1	5	6	6	6	
S2	1	1	5	6	7	7	7
S3	1	1	5	5			
S4	1	1	2	4			
S5	1	1	2	2	3	3	3

Challenge: Log Inconsistencies

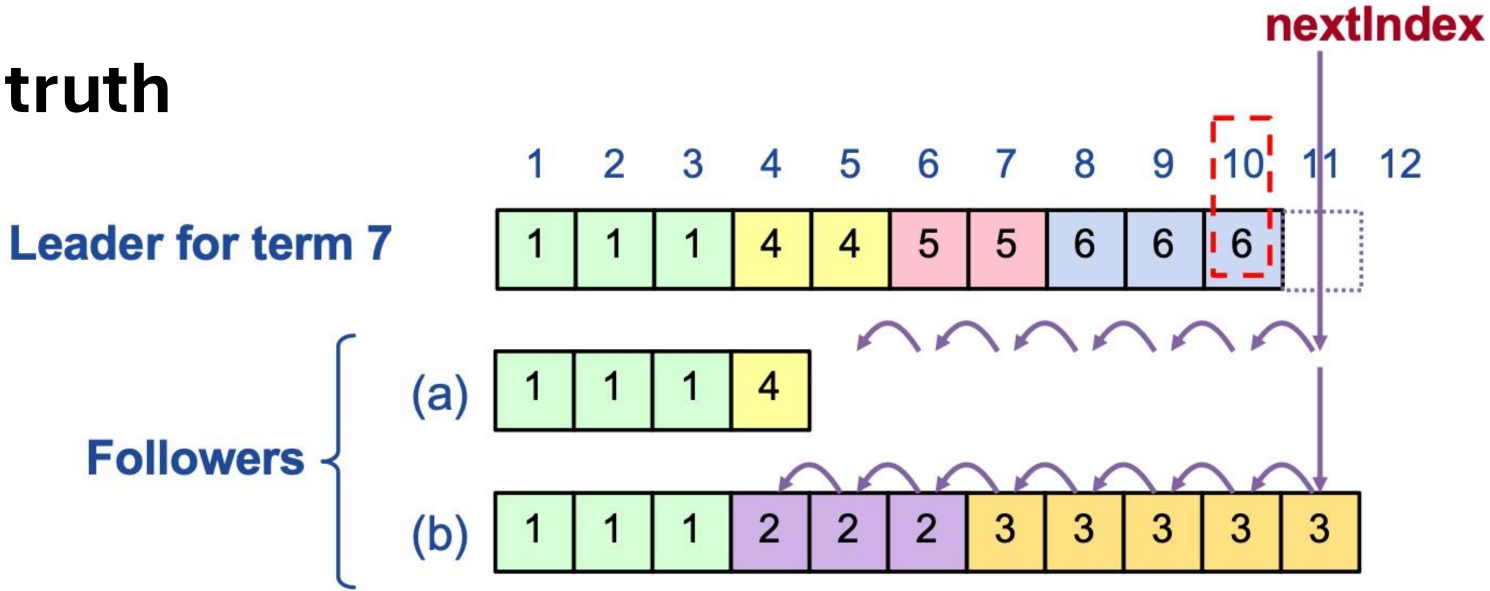


	1	2	3	4	5	6	7	8	9	10	11	12
Leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

Leader changes can result in log in consistencies

Repairing Follower Logs

New leader's log is truth



	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		
(a)	1	1	1	4								
(b)	1	1	1	2	2	2	3	3	3	3	3	

New leader must make follower logs consistent with its own

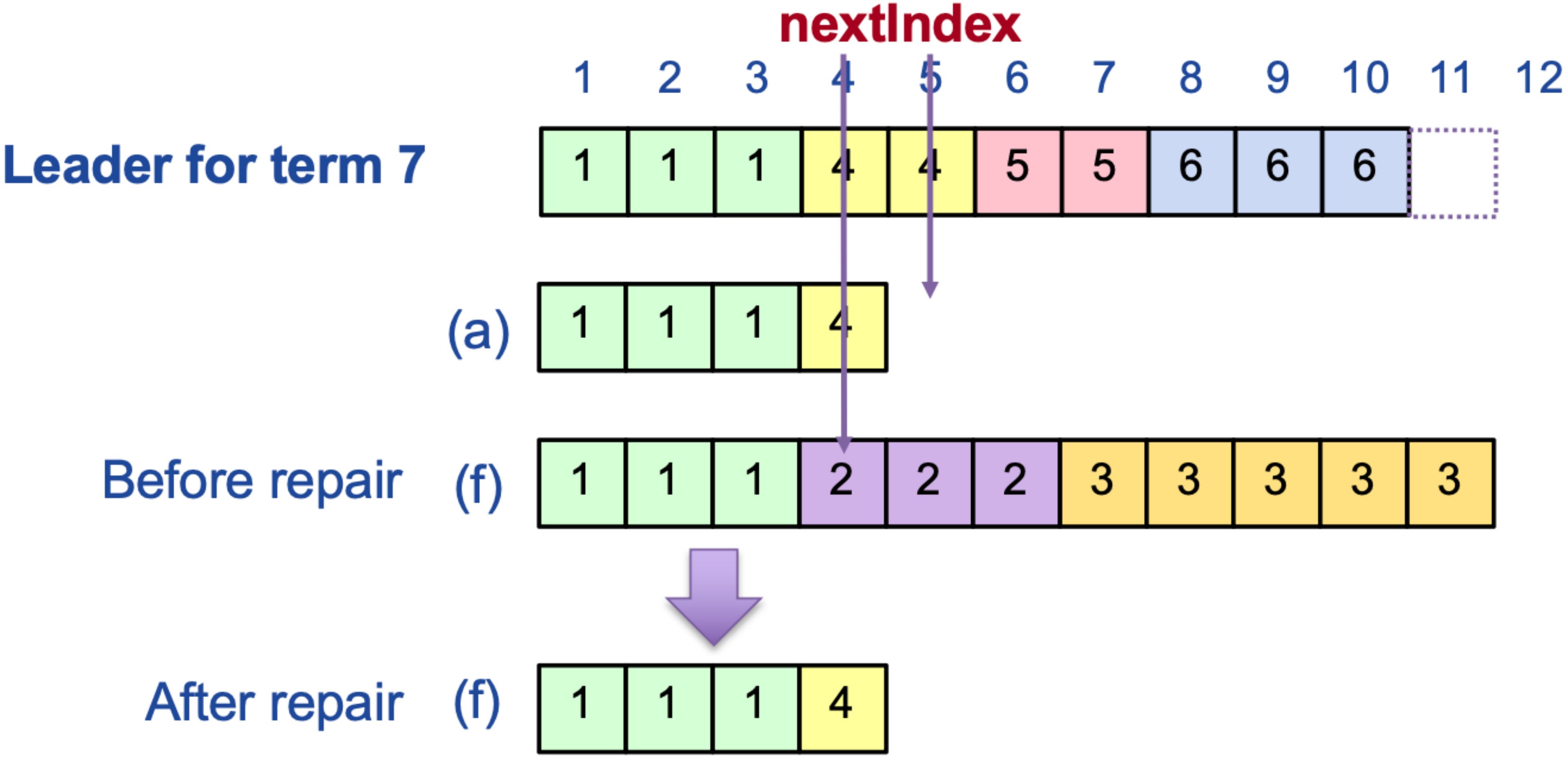
- Delete extraneous entries
- Fill in missing entries

Leader keeps nextIndex for each follower:

- Index of next log entry to send to that follower
- Initialized to (1 + leader's last index)

If AppendEntries consistency check fails, decrement nextIndex, try again

Repairing Follower Logs



Restriction Leader Election

Why not elect the server with the longest log as leader?

- Example

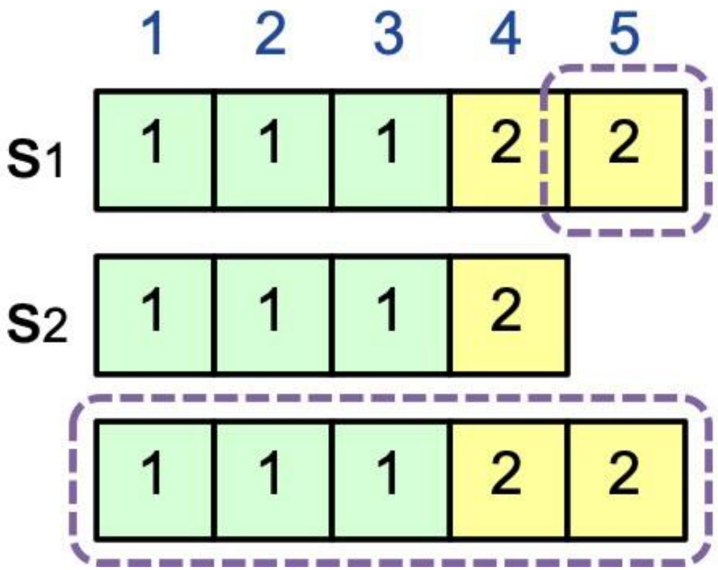
  S1: 5 6 7

  S2: 5 8

  S3: 5 8

Picking The Best Leader

Can't tell which entries committed!



	1	2	3	4	5
S1	1	1	1	2	2
S2	1	1	1	2	
	1	1	1	2	2

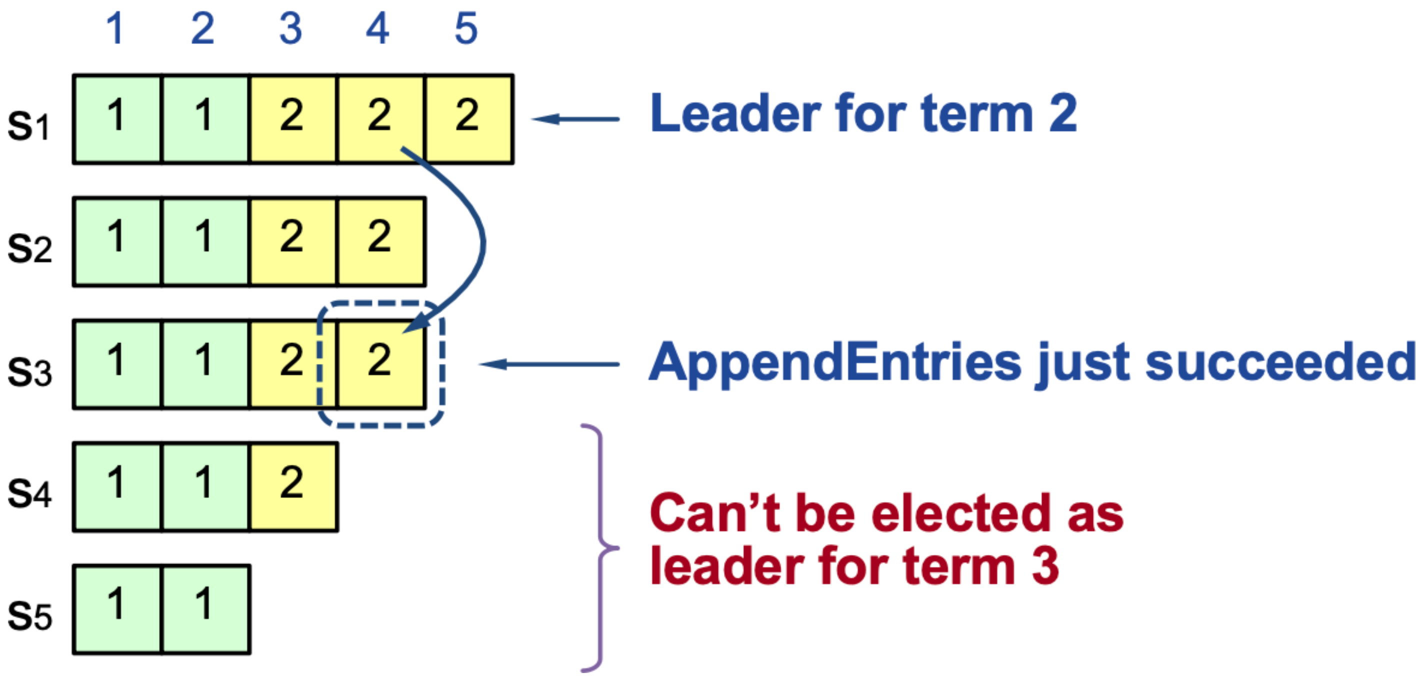
Committed?

Unavailable during leader transition

Elect candidate most likely to contain all committed entries

- In RequestVote, candidates incl. index + term of last log entry
- Voter V denies vote if its log is "more complete": (newer term) or (entry in higher index of same term)
- Leader will have "most complete" log among electing majority

Committing Entry From Current Term



	1	2	3	4	5
S1	1	1	2	2	2
S2	1	1	2	2	
S3	1	1	2	2	
S4	1	1	2		
S5	1	1			

Leader for term 2

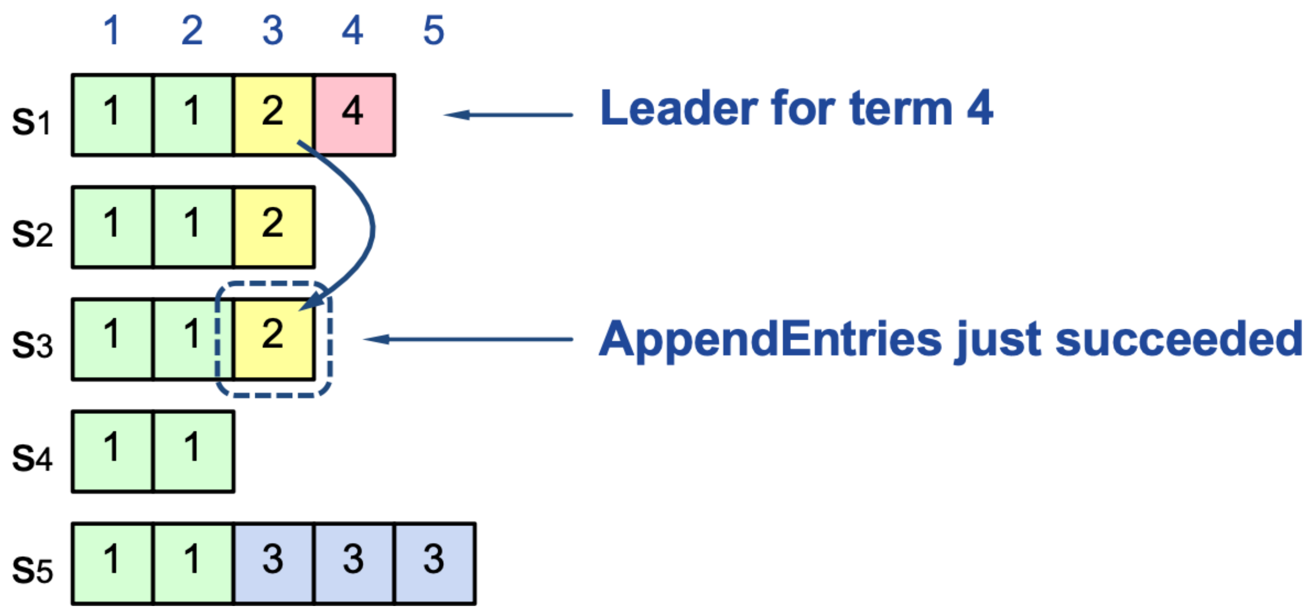
AppendEntries just succeeded

Can't be elected as
leader for term 3

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

Safe: leader for term 3 must contain entry 4

Committing Entry From Earlier Term



	1	2	3	4	5
s1	1	1	2	4	
s2	1	1	2		
s3	1	1	2		
s4	1	1			
s5	1	1	3	3	3

Leader for term 4

AppendEntries just succeeded

Case #2: Leader trying to finish committing entry from earlier

Entry 3 not safely committed:

- s5 can be elected as leader for term 5 (how?)
- If elected, it will overwrite entry 3 on s1, s2, and s3

Persistence

What would we like to happen after a server crashes?

- Raft can continue with one missing server

Two strategies:

- replace with a fresh (empty) server?
- reboot crashed server, re-join with state intact, catch up requires state that persists across crashes

Persistent State

Three variables:

- log[], currentTerm, votedFor

Why log[]?

Why votedFor?

Why currentTerm?

Some Raft state is volatile

- commitIndex, lastApplied, next/matchIndex[]
- why is it OK not to save these?

Persistence is often the bottleneck

A hard disk write takes 10 ms, SSD write takes 0.1 ms

- disk, SSD, battery-backed RAM, &c

Lots of tricks to cope with slowness of persistence:

- batch many new log entries per disk write
- persist to battery-backed RAM, not disk

Log Compaction

How does the service recover its state after a crash+reboot?

- easy approach: start with empty state, re-play Raft's entire persisted log
- problem:
  - log will get to be huge
  - re-play will be too slow for a long-lived system

Insight:

- clients only see the state, not the log
- Service state usually much smaller, so let's keep just that

Solution: ”SnapShot”

- service periodically creates persistent "snapshot”
- service writes snapshot to persistent storage
- service tells Raft it is snapshotted
- Raft discards log before that index

Log Compaction

What happens on crash+restart?

- service reads snapshot from disk
- Raft reads persisted log from disk
- service tells Raft to set lastApplied to last included index to avoid re-applying already-applied log entries

Problem: What if follower's log ends before leader's log starts?

- leader can't repair that follower with AppendEntries RPCs
- InstallSnapshot RPC

Quiz