

CS 425 / ECE 428

Distributed Systems

Fall 2026

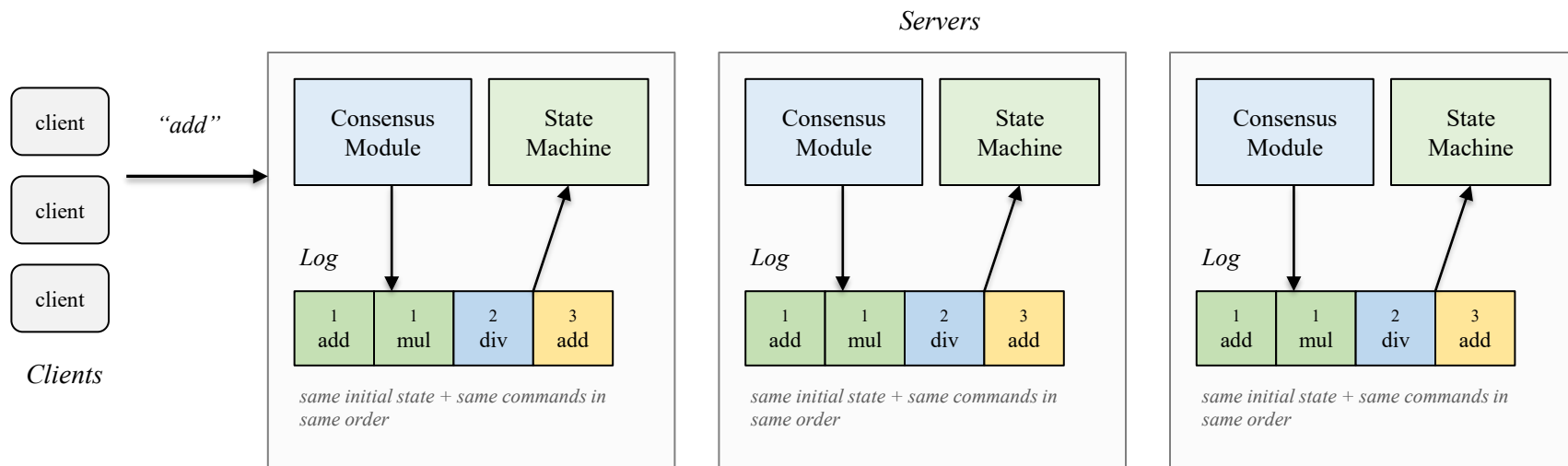
Aishwarya Ganesan w/ Ram Kesavan

Lecture 18-B: Raft

Consensus and Single-decree Paxos

- All processes decide same (some) **single value**
- **For practical applications:**
 - Need to agree on a sequence of values
 - Usually achieved with a replicated log

Idea: State Machine Replication



Each server starts from the same initial state and executes same commands in same order

→ final states and outputs are identical

Consensus module ensures proper log replication

Problem: how to agree on the log?

- **Safety** (recall: what is the safety condition of Paxos?)
 - If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.
- **Liveness**
 - Available as long as any majority of the servers are operational and can communicate with each other and with clients.
- **Failure model: fail-stop, delayed/lost messages**
 - recover from state on stable storage and join the cluster

Raft

- Raft is a consensus algorithm for agreeing on replicated log
- **Elect a leader** to drive log consensus
 - Decomposes the problem (normal operation, leader changes)
 - Simplifies normal operation
 - Eliminates conflicts between proposers
- **Impact:** many implementations, widely adopted
 - etcd (Kubernetes), Consul, CockroachDB, TiKV, RethinkDB

Basics 1: Server States

At any given time, each server is either:

- **Leader:**

- handles all client interactions and log replication
 - at most 1 viable leader at a time (what is “time” here? – later)

- **Follower:**

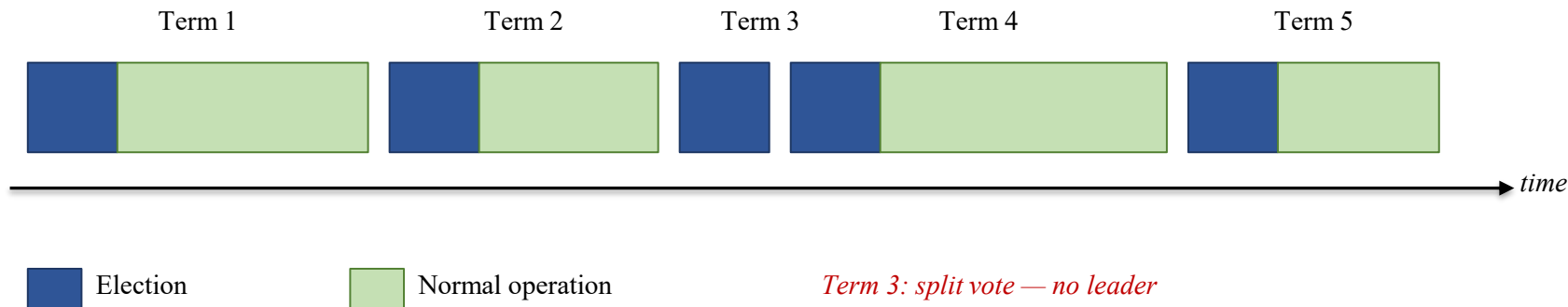
- completely passive
 - only responds to messages from leader/candidates

- **Candidate:**

- temporary state during leader election
 - later becomes leader or follower

Normal operation: 1 leader, $N-1$ followers

Basics 2: Terms



Time is divided into terms. Each term begins with a leader election (how? – later)

Then normal operation under a single leader, until that leader fails

At most 1 leader per term (“at most” and not “exactly one”)

Terms (contd.)

- Each server maintains its current term: its best guess of the current term of the leader
- Terms are monotonically increasing and never reused
- Current term is persisted on disk
- **Key role of terms: identify obsolete information**
 - S1 is in term 2, S2 is in term 4 $\Rightarrow$ S2 knows that S1 is in an older state

Basics 3: The Log

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

Each server has its own copy of the log

A log consists of entries.

Each entry has:

- an index: its position in the log
- a command: can be anything the state machine allows
- a term: the leader's term in which the entry was created

Log is stored on stable storage (disk); it must survive crashes

Normal Operation (no failures)

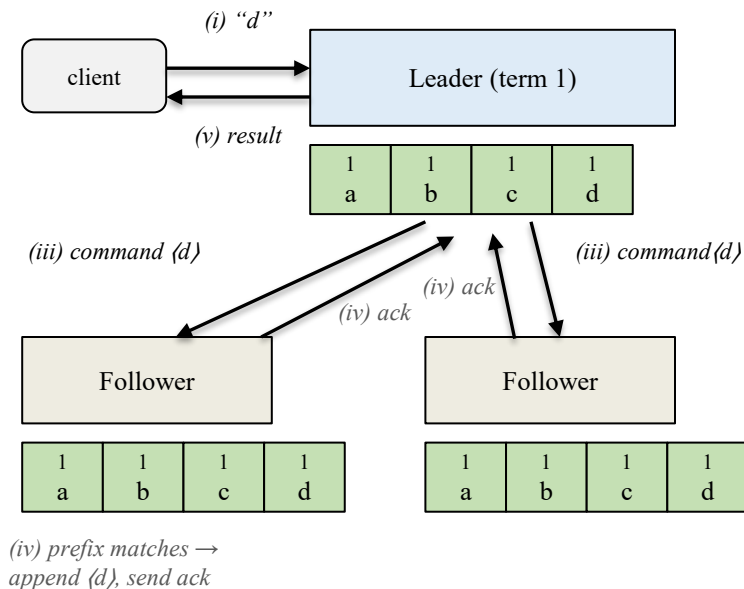
Assume a leader has been elected and all followers are up:

- i. Client sends command to leader
- ii. Leader appends command to its log at index i , in its current term t
- iii. Leader sends command to followers (using AppendEntries RPCs)
- iv. Follower appends command at index i if the prefix of its log matches the leader's (how check? What if not matching? *catching up followers*; later)
- v. Once leader receives acks from a majority (including itself), leader applies the command to its state machine.

entry is **committed** now;

leader returns the result to client

- vi. Leader notifies followers of committed entries in subsequent RPCs



Normal Operation (contd.)

- **Performance in normal case:** one round of messages to majority
- Normal operation ensures identical logs
- **But we must deal with failures!**

CS 425 / ECE 428

Distributed Systems

Fall 2026

Aishwarya Ganesan w/ Ram Kesavan

Lecture 18-B: Raft – Handling failures

Ack: figures adapted from Ongaro & Ousterhout Raft paper and user-study slides

Next

- The leader crashes. How do followers detect it, and how do they elect a new leader?

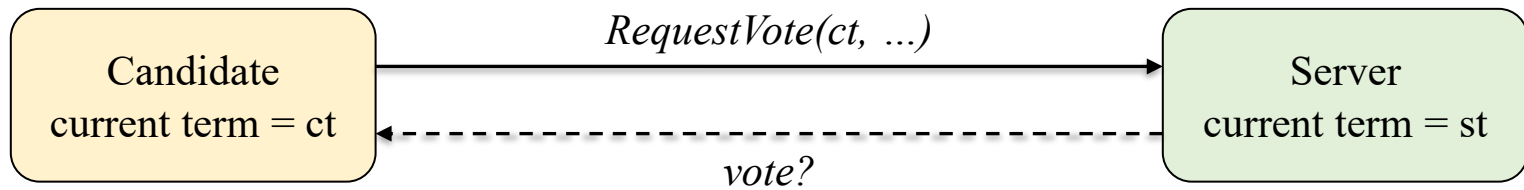
Heartbeats and Timeouts

- Leader must send **periodic heartbeats** to maintain authority
- Followers expect to receive RPCs from a leader (or candidates)
- If a period of time (*electionTimeout*) elapses with no RPCs:
 - Follower assumes the leader has crashed (or partitioned)
 - Leader could have been slow
 - Follower cannot tell the difference — recall failure detectors!
 - Follower starts a new election
 - Timeouts typically 100–500 ms

Election Basics

- Follower converts to the **candidate** state and runs for election
- **Increments its current term** before running — why?
 - There can be only one leader per term, and we already had a leader for this term (which crashed)
 - Raft would never let the candidate win the old term; no point running in it
- Votes for itself
- Requests for votes from all other servers (using RequestVote RPCs)
- Waits for votes from a majority
- **How does a server decide whether or not to give its vote to a candidate?**

Voting Conditions



A server with current term st **denies** its vote to a candidate with term ct if:

(i) the server has **already voted for** a different candidate in its current term st

Why? We cannot have two leaders for the same term.

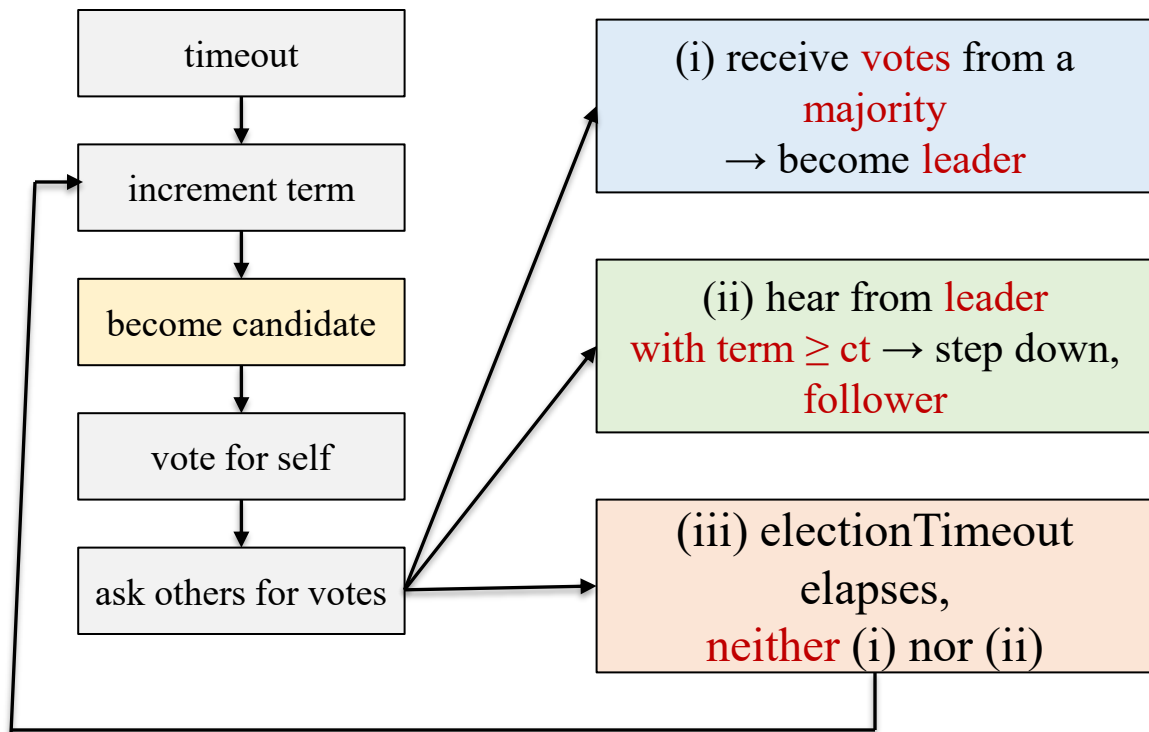
How does the server ensure it never votes twice in a term?

remembers the candidate id server voted for in the current term on disk

(ii) $ct < st$: the **candidate is in an older term**

(iii) the candidate's **log is lagging** (defined precisely later)

Election: Candidate Outcomes



Once a node becomes leader, what must it do to maintain authority?
→ send heartbeats to all other servers

Split votes: nobody wins before the timeout
→ try again

repeat: increment term, new election

Elections: Safety

Election safety: allow at most one winner per term

- Each server gives out only one vote per term
- Two different candidates cannot both accumulate majorities in the same term

How about liveness?

Is it possible to end up in a term with no leader?

Elections: Liveness

- **Split votes:** leader for term t fails; all followers time out at the same time
 - Each votes for itself in term $t+1$ → no node gets a majority
 - Timeout... all nodes vote for themselves again in term $t+2$. Repeat...
- **How to ensure liveness (no indefinite split votes)?**
 - Randomness: each server chooses its electionTimeout uniformly at random in $[T, 2T]$
 - One server usually times out and wins the election before others
 - $T \gg$ broadcast time (time to send an RPC to every server and get responses)
- **What if T is too high? Too low?**

Next

- How does a new leader repair/catch up follower logs?

Log Properties

- Log entries have **terms** that are **monotonically increasing**
- If entries on different servers have the **same index and term**:
 - they store the **same command**
 - the logs are **identical** in all **preceding** entries

(Log Matching property)
- If a given entry is **committed**, all **preceding entries** are also committed.

Catching up Followers' Logs

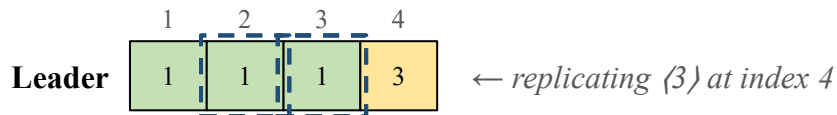
- Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous entries
- Leader eventually makes followers' logs identical to its own.
 - delete extraneous entries, fill in missing entries

Consistency Check During Log Appends

- Leader keeps **index of the next entry** to send to a follower (nextIndex); initialized to (1 + leader's last index)
- AppendEntries RPC contains the index and term of the **preceding entry** (prevLogIndex, prevLogTerm)
- If follower does **not** contain a matching preceding entry, decrement nextIndex, and try again
- When it **matches**, follower accepts the new entry after truncating any conflicting tail
- Implements an **induction** step; ensures Log Matching property.

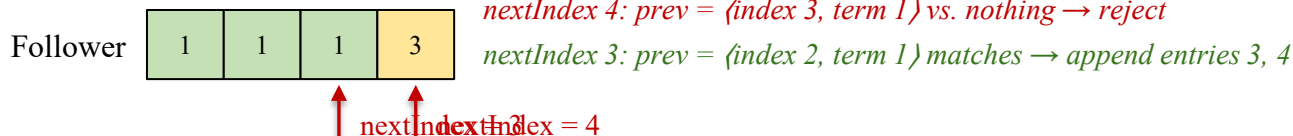
Follower Log Scenarios

Leader for term 3 is currently replicating entry {3} at index 4. `nextIndex` initialized to 4 for follower.



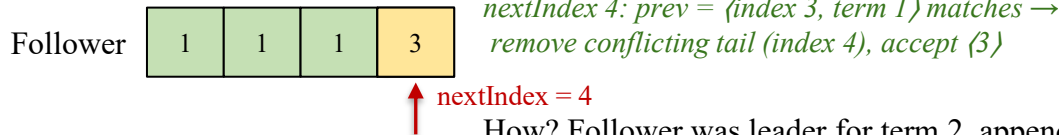
Missing entries

How? Follower crashed before the term-1 leader replicated index 3 to it.



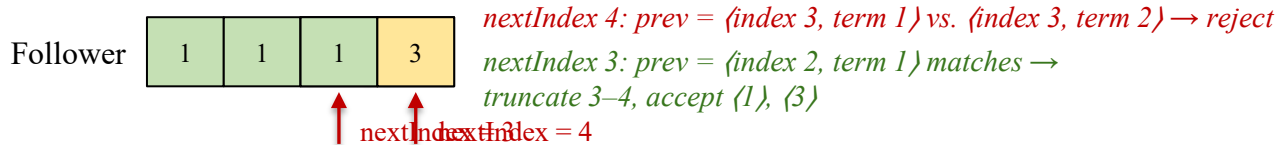
Additional entries

How? Follower was leader for term 1, added {1} at index 4 locally, crashed before replicating.



Missing and additional

How? Follower was leader for term 2, appended two entries, crashed before replicating to a majority.

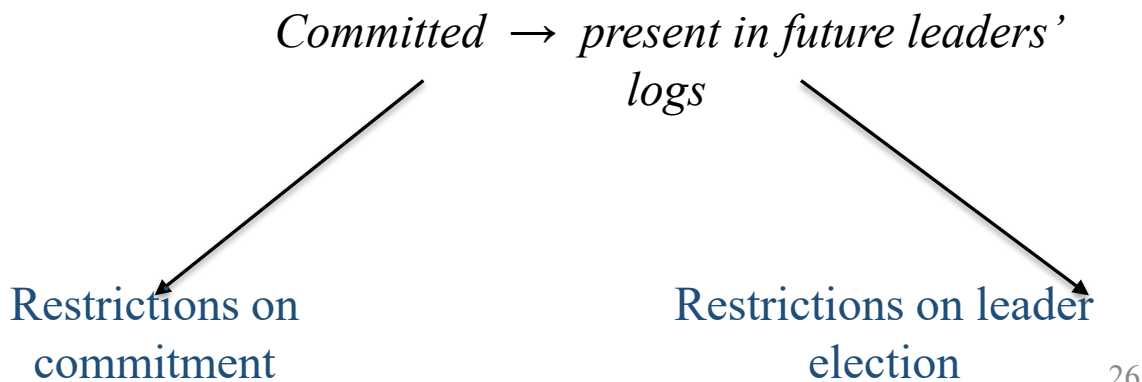


Next

- Followers **blindly trust** the leader. So which candidate is it safe to elect?

Safety Requirement

Leader Completeness: if a log **entry** is **committed** in a given **term**, then that entry will be **present** in the logs of the **leaders** for all higher-numbered **terms**.



Why Leader Completeness Matters

Term 1: S1 is leader, replicates entries 1–3 on S1 and S2 (a majority), applies them to its state machine, then fails.

	1	2	3
S1	1	1	1
S2	1	1	1
S3	1	1	

S3 has a stale log

*suppose S3
incorrectly
wins term 2*



	1	2	3
S1	1	1	2
S2	1	1	2
S3	1	1	2

*S3 (term 2) replicates its own entry
at index 3*

Entry {1} at index 3 was
committed and applied by S1,
then overwritten — lost!

Wrong!

Restrictions on Leader Election

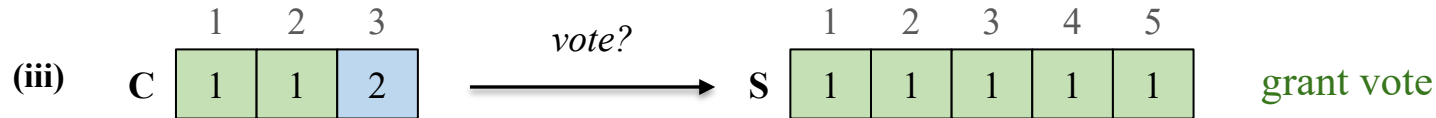
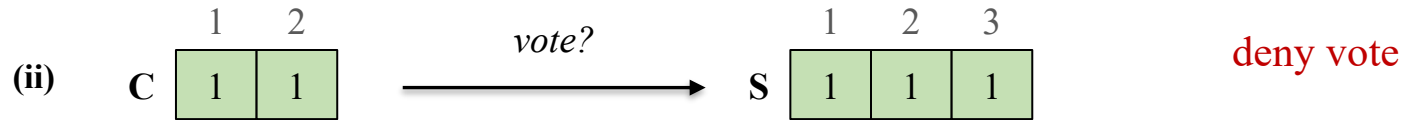
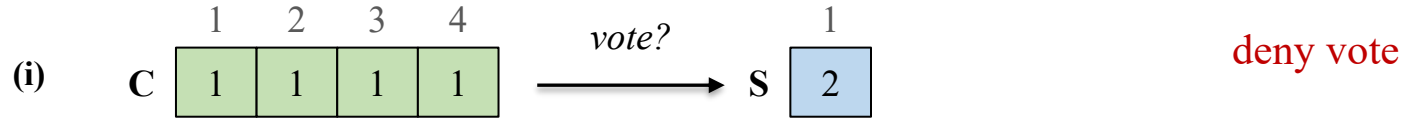
- During elections, choose the **candidate C with log containing all committed entries**
- Candidates include log info in RequestVote RPCs: index and term of their last log entry
- Voting server V **denies** its vote if its **own log is “more complete”**:

$$\begin{aligned} &(\text{lastTermV} > \text{lastTermC}) \parallel \\ &(\text{lastTermV} == \text{lastTermC}) \ \&\& \ (\text{lastIndexV} > \text{lastIndexC}) \end{aligned}$$

Leader will have the “**most complete**” log among the electing majority

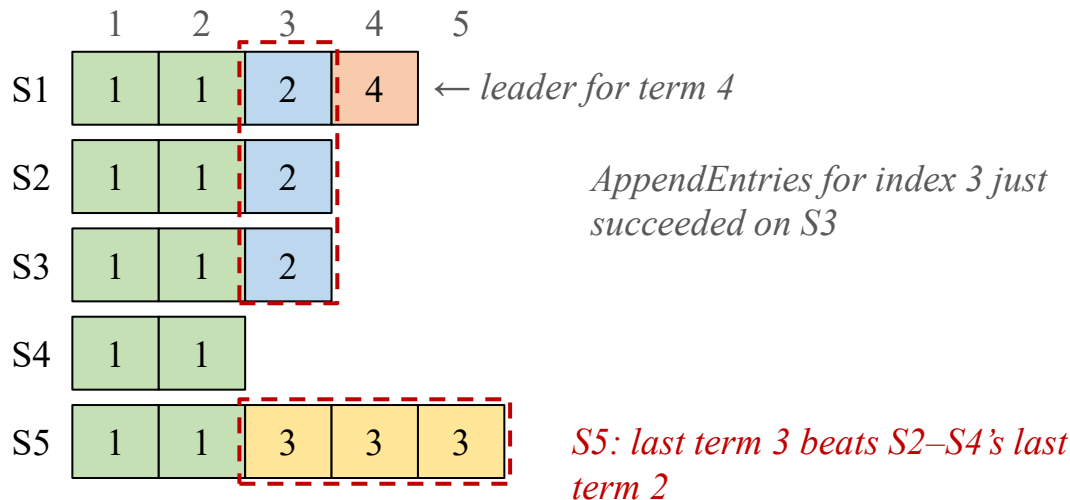
Which log is more complete?

Candidate C asks server S for a vote. Does S grant it?



Committing: Entry from Earlier Term

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



Entry 3 is now on a majority (S1, S2, S3); but it **cannot be safely committed**:

S5 can still be elected leader for term 5

(its last term $3 > 2$, so S2, S3, S4 would vote for it).

If elected, it overwrites entry 3 on S1, S2 and S3!

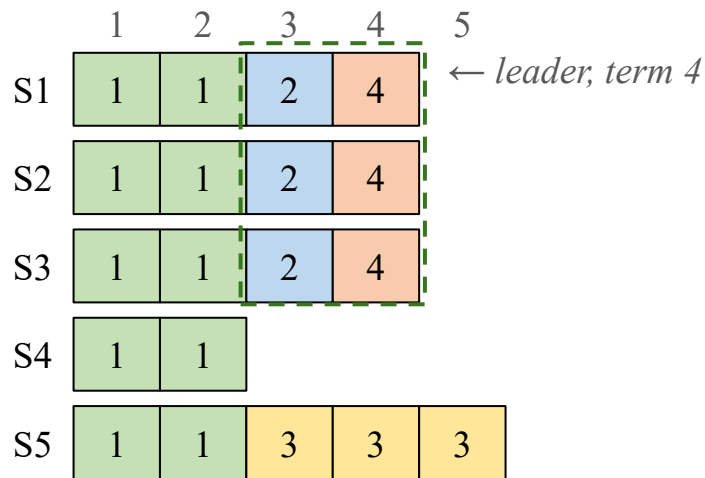
Restrictions on Commitment

For a leader to decide an entry is **committed**:

- that **entry** must be on a **majority**
- At least one entry from **leader's current term** must **also** be on a majority

Once entry 4 (term 4) is committed:

- S5 cannot be elected leader for term 5
- Entries 3 and 4 are both safe



Safety: Intuition

- **Only committed entries are applied**
 - committed: on a majority, including an entry from current leader's term
- **Any leader elected later must have won votes from a majority**
 - that majority overlaps the one holding the committed entry
 - some voter V has it.
- **V voted only because candidate's log was at least as up-to-date**
 - by Log Matching the new leader holds everything V holds, including the committed entry $\Rightarrow$ Leader Completeness.
- **Leaders never overwrite their own log**
 - so once an entry is committed at index i no server can ever apply anything else at i $\Rightarrow$ State Machine Safety.

Wrap Up

- **Reading:** Ongaro & Ousterhout, “In Search of an Understandable Consensus Algorithm”
- <https://raft.github.io/raft.pdf> (Sections 2 and 5)