



# RSM & PAXOS & Raft **Consistency across replicas**

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS & Jinyang Li@NYU & John Ousterhout

# **Pessimistic Replication**

## Some applications may prefer not to tolerate inconsistency

- E.g., a replicated lock server, or replicated coordinator for 2PC
  - Better not give out the same lock twice
- E.g., Better have a consistent decision about whether transaction commits

## **Trade-off: stronger consistency with pessimistic replication means:**

- Lower availability than what you might get with optimistic replication
- Performance overhead for waiting syncing w/ other replicas

# **Review: Single-copy Consistency (Linearizability)**

## Problem of optimistic way: replicas get out of sync

- One replica writes data, another doesn't see the changes
- This behavior was impossible with a single server

## Ideal goal: single-copy consistency

- Property of the externally-visible behavior of a replicated system
- Operations appear to execute as if there's only a single copy of the data
  - Internally, there may be failures or disagreement, which we have to mask
- Similar to how we defined serializability goal ("as if executed serially")

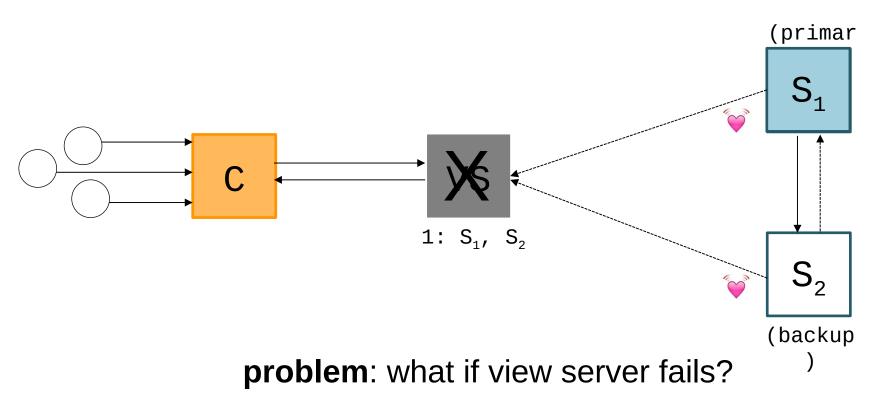
# **Review: Replicated State Machines**

#### A general approach to making consistent replicas of a server:

- Start with the same initial state on each server.
- Provide each replica with the same input operations, in same order
- Ensure all operations are deterministic
  - E.g., no randomness, no reading of current time, etc.

These rules ensure each server will end up in the same final state

# Review: primary/backup model & view server



Now, we need **Paxos** 

# Quorum

## Goal: assume a single write finishes its writes

A later reader will read its value

#### Write execution

The writer sends requests to Nreplicas, return ok if Qw servers ACK the write

#### Read execution

The reader sends requests to Nreplicas, return ok if Qr servers ACK the read

## Quorum: Qr + Qw > Nreplicas (Can achieve our goal; why?)

- Confirm a write after writing to at least Qw of replicas
- Read at least Qr agree on the data or witness value

# **Review: Majority in Distributed Systems**

## Why does the majority rule work?

- Any two majority sets of servers overlap
- Suppose two clients issue operations to a majority of servers
- Must have overlapped in at least one server, will help ensure single-copy

## When is it OK to reply to client?

- Must wait for majority of replicas to reply
- Otherwise, if a minority crashes, remaining servers may continue without op

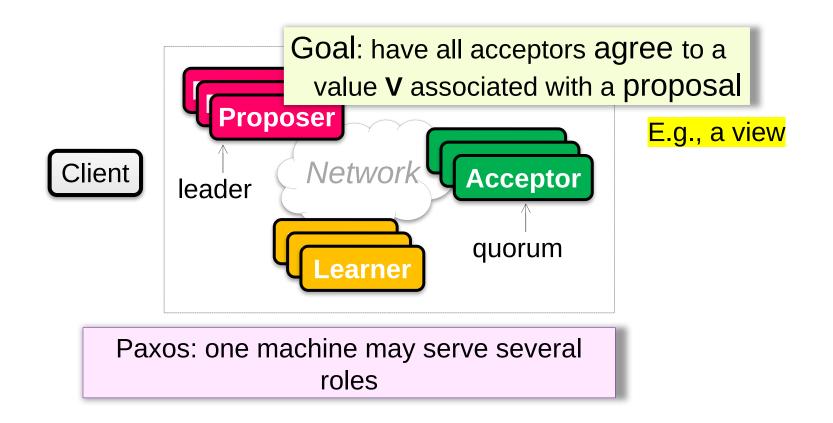
## Remaining question: what if multiple writers exist?

# **Single-decree Paxos**

Agree on a single value

# Paxos' properties: correct + fault-tolerance

No guaranteed termination (i.e., lack of availability guarantee)



# **Paxos Players**

Client

makes a request

**Proposer** 

Get a request and run the protocol **Leader** = elected Coordinator

**Acceptor** 

Remember the state of the protocol **Quorum** = any majority of Acceptors

Learner

When agreement has been reached, a Learner executes the request and/or sends a response back to the Client

# **Review: Political Science 101 in Paxos**

## Paxos has rounds; each round has a unique ID (N, i.e., proposal number)

Goal: "the propose with the highest proposed number win"

## **Rounds are asynchronous**

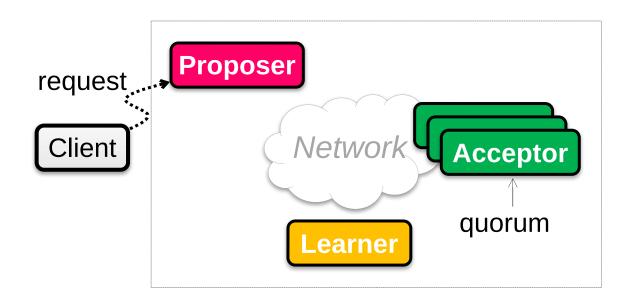
- Time synchronization not required
- If you are in round j and hear a message from round j+1, abort everything and move over to round j+1
- Use timeouts; may be pessimistic

## Each round itself broken into phases

Phases are also asynchronous

# Paxos in Action: Phase 0

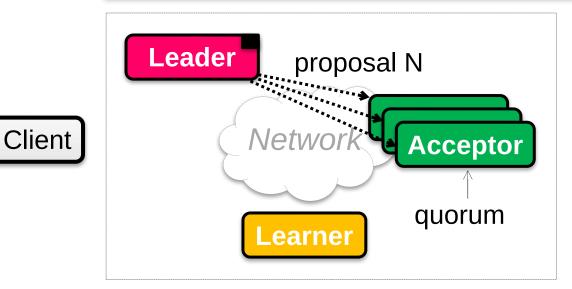
# Client sends a request to a proposer



# Paxos in Action: Phase 1a (Prepare)

# Leader creates a proposal N and send to quorum

**N** is greater than **any** previous proposal number seen by this proposer

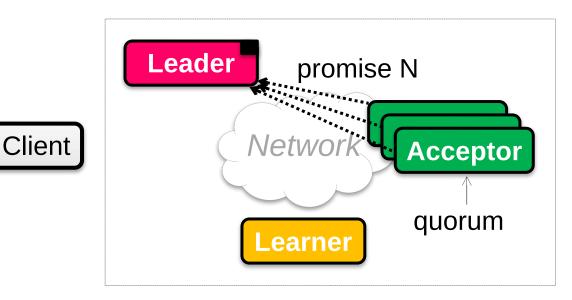


# Paxos in Action: Phase 1b (Prepare)

Acceptor: if proposal ID > any previous proposal

- 1. reply with the highest past proposal number and value
- 2. promise to ignore all IDs < N

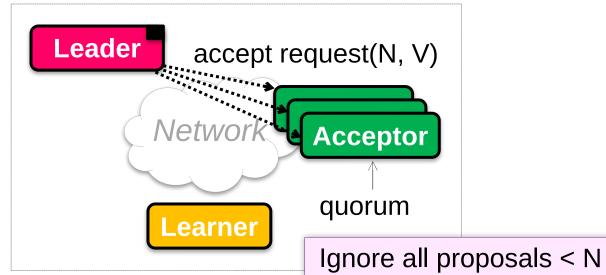
else ignore (proposal is rejected)



# Paxos in Action: Phase 2a (Accept)

**Leader: if receive enough promise** 

- 1. set a value V to the proposal V, if any accepted value returned, replace V with the returned one
- 2. send accept request to quorum with the chosen value V



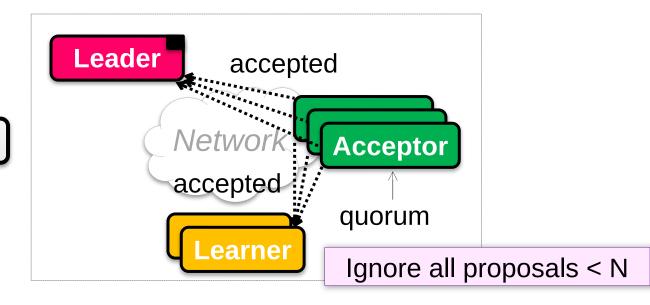
Client

# Paxos in Action: Phase 2b (Accept)

**Acceptor:** if the promise still holds

- 1. register the value V
- 2. send accepted message to Proposer/Learners

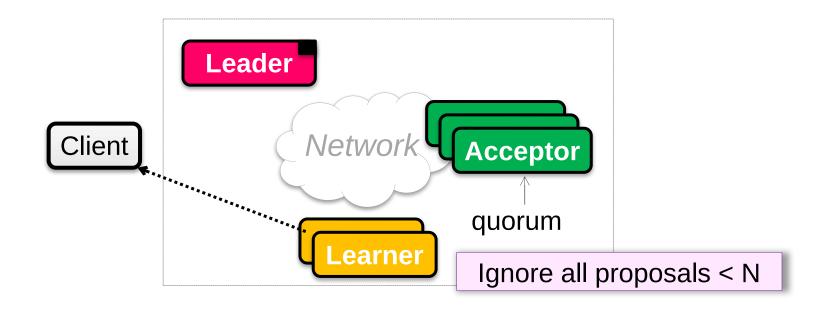
else ignore the message

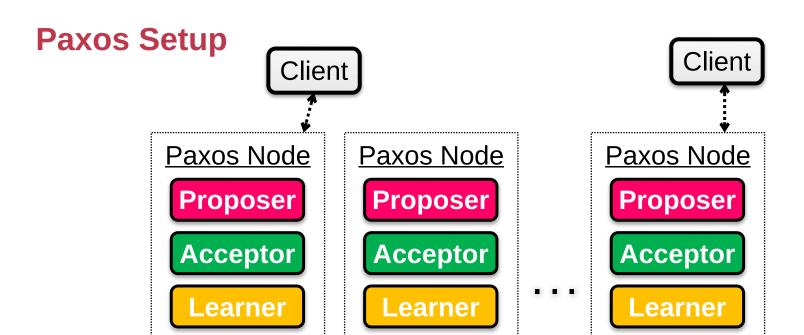




# Paxos in Action: Phase 3 (Learn)

# Learner: responds to Client and/or take action on the request





 $N_a$ : highest proposal number accepted

 $V_a$ : accepted value of  $N_a$ 

 $N_h$ : highest proposal number seen

**M**<sub>n</sub>: my proposal number each round of Paxos, each Node

 $N_a$ : highest proposal number accepted

V<sub>a</sub>: accepted value of N<sub>a</sub>

 $N_h$ : highest proposal number seen

 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

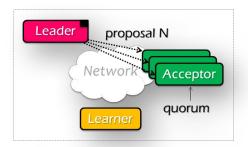
#### A node decides to be Leader

Leader chooses  $M_n > N_h$ 

Leader sends <proposal,  $M_n >$  to all nodes

# Acceptor receives ceives, N>

```
if N < N_h reply reply reject>
else
N_h = N
reply reply Proposal N
reply r
```



 $N_a$ : highest proposal number accepted

V<sub>a</sub>: accepted value of N<sub>a</sub>

 $N_h$ : highest proposal number seen

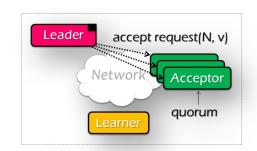
 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

#### If Leader gets promise-ok from a majority

```
if V := null, V = the value of the highest <math>N_a received if V = null, then Leader can pick any V send <accept, M_n, V > to all nodes
```

# If Leader fails to get majority promise-ok delay and restart Paxos

```
Upon receiving <accept, N, V>
   if N < N<sub>h</sub>
     reply <accept-reject>
   else
     N<sub>a</sub> = N; V<sub>a</sub> = V; N<sub>h</sub> = N;
   reply <accept-ok>
```



 $N_a$ : highest proposal number accepted

 $V_a$ : accepted value of  $N_a$ 

 $N_h$ : highest proposal number seen

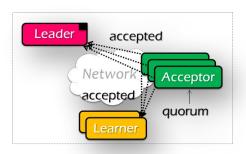
 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

# If Leader gets accept-ok from a majority

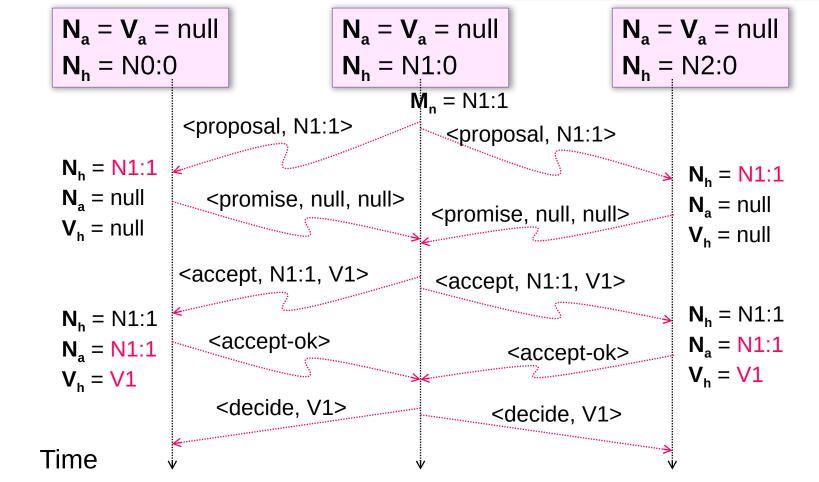
send <decide,  $V_a$ > to all nodes

# If Leader fails to get majority accept-ok

delay and restart Paxos



 $N_a$ : highest proposal number accepted  $V_a$ : accepted value of  $N_a$   $N_h$ : highest proposal number seen  $M_n$ : my proposal number





 $N_a$ : highest proposal number accepted

V<sub>a</sub>: accepted value of N<sub>a</sub>

**N**<sub>h</sub>: highest proposal number seen

 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

## Why setups multiple acceptors?

Failure of the single acceptor halts decision

## Why not accepts the first proposal and rejects the rest?

- Leader dies
- Multiple leaders result in no majority accepting

#### What if more than one leader is active?

Can both leaders see a majority of promises?

# **Inside of Paxos**

N<sub>a</sub>: highest proposal number accepted

V<sub>a</sub>: accepted value of N<sub>a</sub>

 $N_h$ : highest proposal number seen

 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

#### When is the value V chosen?

- ☐ Leader receives a majority <promise, ...>
- □ A majority acceptors receive <accept, N, V>
- □ Leader receives a majority <accepted, ...>

#### What if acceptor fails after sending promise?

Must remember N<sub>h</sub>

#### What if acceptor fails after receiving accept?

 $\square$  Must remember  $N_h$  and  $N_a V_a$ 

## What if leader fails while sending accept?

Propose M<sub>n</sub> again

 $N_a$ : highest proposal number accepted  $V_a$ : accepted value of  $N_a$   $N_h$ : highest proposal number seen  $M_n$ : my proposal number

Suppose that the acceptors are A, B, and C. A and B are also proposers. How does Paxos ensure that the following sequence of events can't happen? What actually happens, and which value is ultimately chosen?

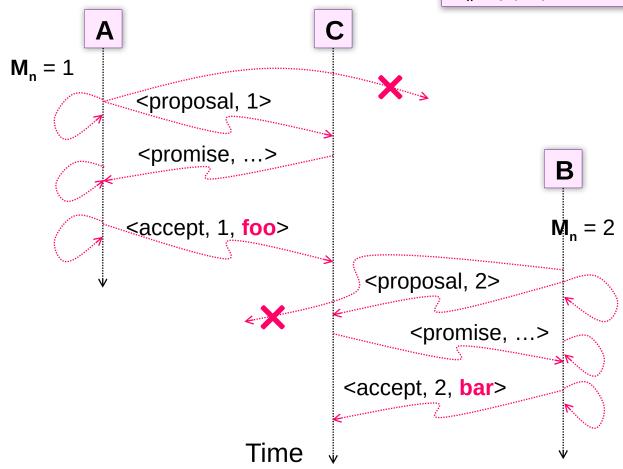
- A sends <accept,1, "foo"> to A and C and gets responses from both. Because a majority accepted, A thinks that "foo" has been chosen. However, A crashes before sending an <accept, 1, "foo"> to B
- B sends <accept, 2, "bar"> messages to B and C and gets responses from both, so B thinks that "bar" has been chosen

 $N_a$ : highest proposal number accepted

 $V_a$ : accepted value of  $N_a$ 

 $N_h$ : highest proposal number seen

 $\mathbf{M}_{\mathbf{n}}$ : my proposal number

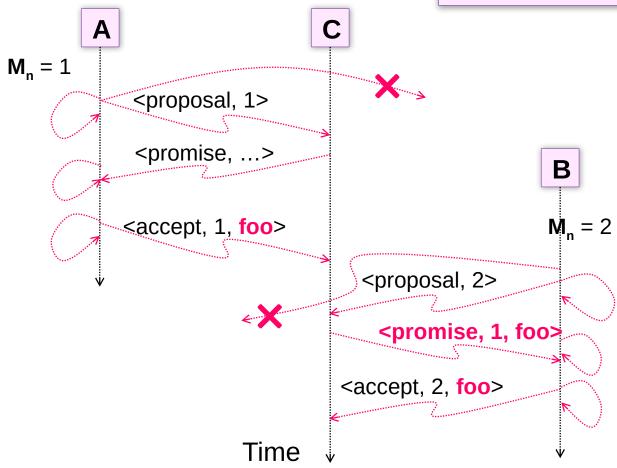


 $\mathbf{N}_{\mathbf{a}}$ : highest proposal number accepted

 $V_a$ : accepted value of  $N_a$ 

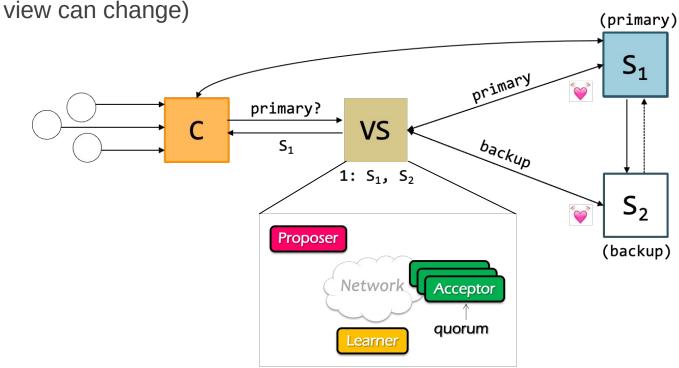
 $N_h$ : highest proposal number seen

 $\mathbf{M}_{\mathbf{n}}$ : my proposal number



## Does single-decree Paxos works fine for our view server example?

When can it go wrong? Accepting a single value is not enough (because view can change)



# Multi-Paxos builds on top of the basic Paxos

## Useful when agreeing on a sequences of values, examples including:

- Views in primary-backup replication
- Logs in a replicated state machine
  - i.e., use Multi-Paxos to implement RSM

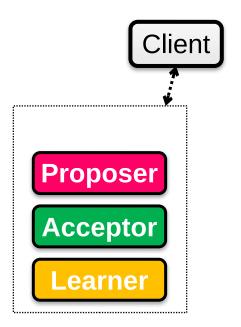
## The basic approach

- Run a separate instance of Paxos to agree on the value of each index
- Each instance of Paxos has its own copy of state
  - highest proposal seen
  - accepted proposal number
  - accepted proposal value

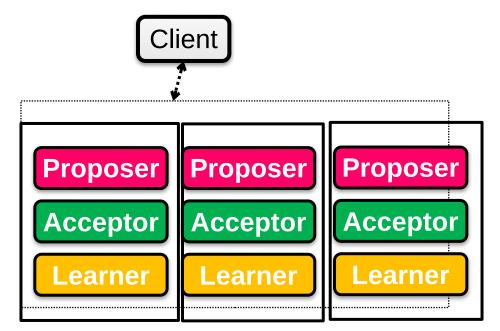
# Single-decree Paxos vs. Multi-Paxos

0 1 2

Log add cmp xxx



**Single-decree Paxos** 



Single-decree instance for 0

Single-decree instance for 1

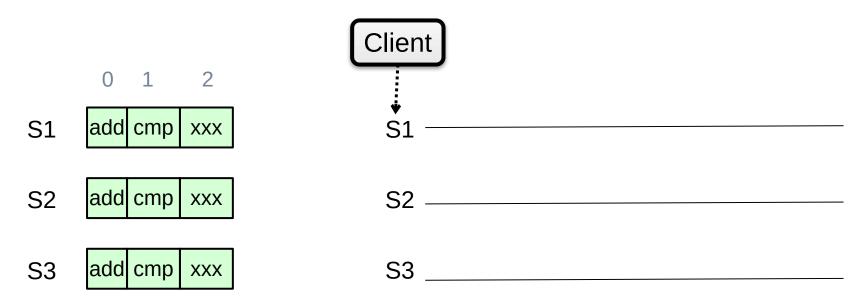
Single-decree instance for 2

**Multi-Paxos** 

## Server simultaneously acts as proposer, acceptor & learner

## **Example**

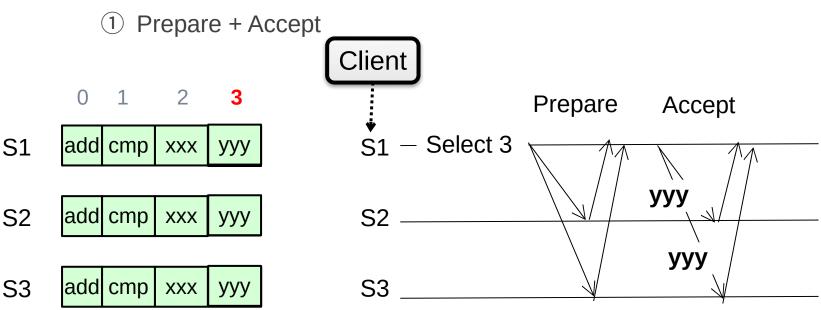
Suppose we want to append an entry to the log



Question: what could happen if instance 3 has already gotten a value?

## After receiving the request from a client, the server will

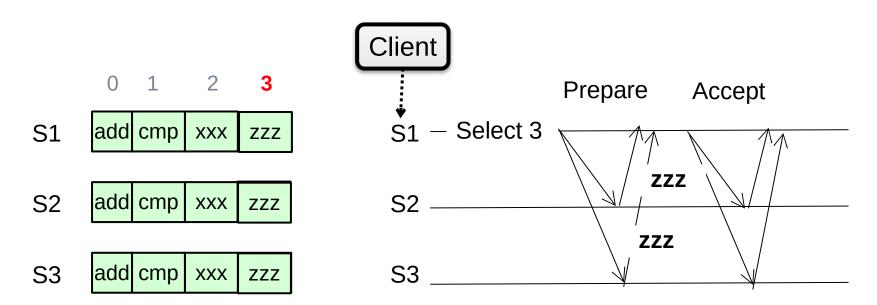
- ① Decide the number of the instance to append (3 in our example)
- ② Do the single-decree paxos to append the log entry to the instance 3



Question: what could happen if instance 3 has already gotten a value?

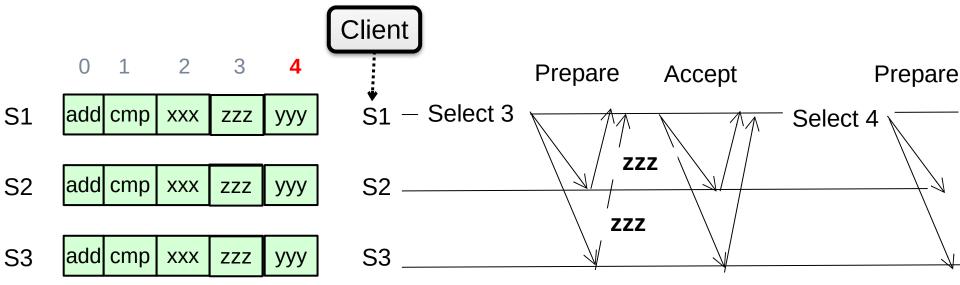
Example: S1 receives the request, but S2 and S3 has already accepted the value (zzz)

Our single-decree paxos will not choose the value client sent



## In case of conflict, S1 will choose a larger instance (e.g., 4)

- And re-run the instance until find the largest available instance
- May possible try many times until find an available slot



How many RTTs are required for appending a single log entry?

## **Basic Multi-Paxos is correct but inefficient**

## 2 rounds of RPCs for each value chosen in the optimal case

- Prepare, Accept

## With multiple concurrent proposers, conflicts and restarts are likely

- 1. The server may conflict on the position of the new log entry to insert (our previous example)
- 2. Even in the same Paxos instance, different proposer may conflict
  - Only the proposers with the highest number can win

#### **Solution**

- ① Select a leader: most of the time, only one server can propose
- 2 Batch the prepare requests from multiple instances sent from a leader

# Multi-paxos uses a distinguished proposer (leader)

# Distinguished proposer (aka. leader)

- The only one that issues proposals
  - i.e., no proposer conflicts in the optimal case

## Client only sends the commands to the leader

Decides the value position

## Note, single leader is *not* necessary for Multi-paxos

 E.g., if two or more servers act as proposers at the same time, the protocol still works correctly

# Not relying a single leader simplifies user implementation

## i.e., the user can choose the leader in an arbitrary way

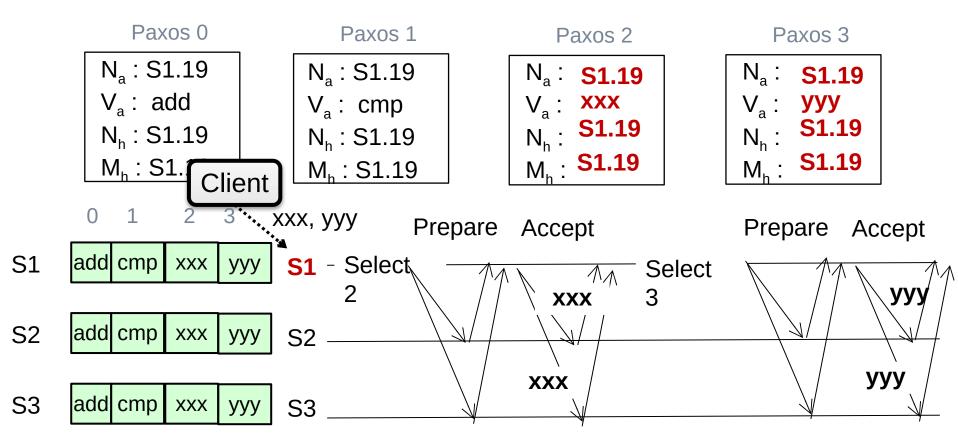
- i.e., all the server don't need to agree on the same leader
- Performance degradations if multiple leaders exist

# We can also use multi-paxos for the leader election ◀

Similar to implementing a view server

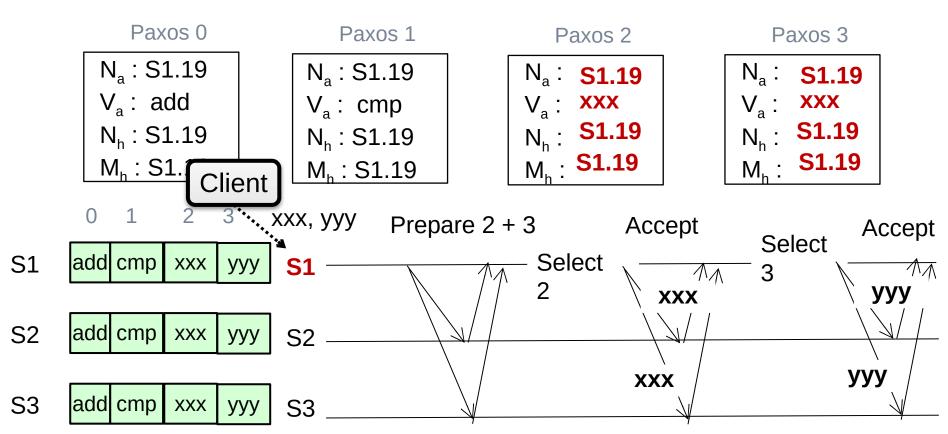
# Benefits of leader election: prepare message batching

# Suppose S1 is the leader (w/o batching)



# Benefits of leader election: prepare message batching

# Suppose S1 is the leader (+ batching)



# Problem: hole in the log

0 1 2 3

Suppose S1 is the leader (with batching)

- S1 add cmp yyy
- What happens after S1 crashes w/ unfinished entry?

Application should fill the gap itself, e.g., w/ no-op

S2 add cmp

Entry 2 will be empty

- add

ууу

# **Summary of Multi-Paxos**

Based on single-decree Paxos, realizing RSM

Needs non-trivial application efforts to make it work

- E.g., should use a leader for better performance
- Should fill the holes of the log entries

# More: Raft replicated log

## A different approach for consensus

- Paxos's 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?

# Raft is described in a concrete form that favors system design & impl.

- Widely implemented and deployed nowadays
- Especially for open-source projects





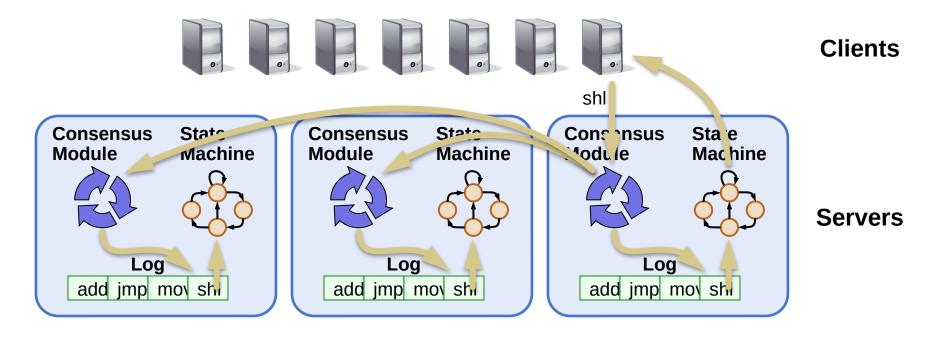








# Abstraction of the operations: replicated log



# Replicated log => replicated state machine

All servers execute same (deterministic) commands in same order

Consensus module ensures proper logs are the same!

# Raft's high-level approach: problem decomposition

#### 1. Leader election

Select one server as the leader

Detect crashes, choose new leader

# 2. Log replication (normal operation)

Leader accepts commands from clients, append to its log

Leader replicates its log to other servers (overwrites inconsistencies)

## 3. Safety

Keep logs consistent

Only servers with up-to-date logs can become the leader

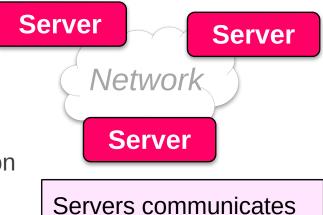
# Raft server states

## At any time, each server is either:

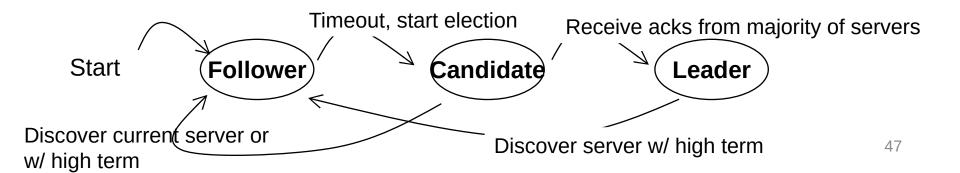
- Leader: handles all client interactions, log replication
  - Invariant: At most 1 viable leader at a time
- Follower: passive (only responds to incoming RPCs)
- Candidate: used to elect a new leader

#### Normal workloads

1 server is the leader, others are the followers



Servers communicates
w/ RPCs



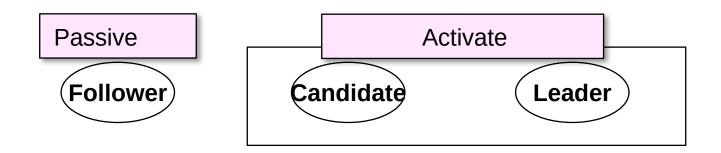
# Raft server states

#### Leader is active

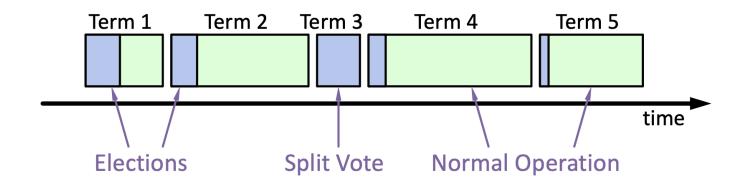
Receive client requests, replicate logs to the followers

# Followers are passive

Only respond to the requests from leaders or candidates



# Raft basics: terms for one leader



# Raft divides time into terms (with arbitrary length):

- Each term starts with an election
- Ends with one leader or no leader
- At most one leader per term

# Each leader is uniquely associated with a term

Key role: identify obsolete information

# Raft server state switch: Heartbeats & Timeouts

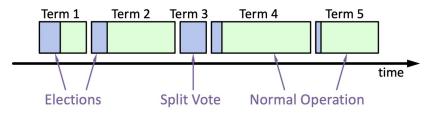
## Servers start as followers (except one server is assigned as the leader)

Followers expect to receive RPCs from leaders or candidates

## Leaders must send heartbeats to maintain authority in a term

- If election timeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

# **Election Basics**



- 1. Change its state to the candidate state
- 2. Increment current term
- 3. Vote for itself

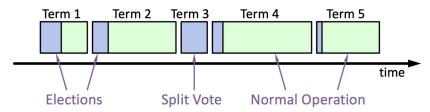
Send RequestVote RPCs to all other servers, retry until either

- ① Receive votes from majority of servers → Become the leader!
- ② Receiver RPC from a valid leader → Return to the follower

Question: what is the condition for the follower to vote?

The leader's term >= self's term & never vote before

# **Election Basics**



- 1. Change its state to the candidate state
- Increment current term
- 3. Vote for itself

Send **RequestVote RPC**s to all other servers, retry until either

- ① Receive votes from majority of servers → Become the leader!
- 2 Receiver RPC from a valid leader > Return to the follower
- ③ No one wins election (timeouts on the election) → Increase the term, start the new election

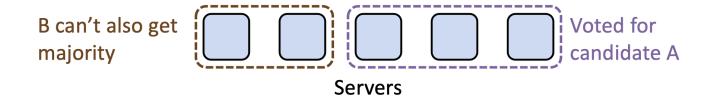
# **Election requirements: safety**

## At most one winner per term

- How to achieve so? Each server only gives one vote per term (persist on disk)
- Majority ensures at least one leader (similar to phase #1 of Paxos)

## Question

- Can different candidates exist in the same term?
- Yes. Each server should keep a VotedFor variable to ensure it only gives vote to one candidate.



# **Election requirements: liveness**

## Some candidate must eventually win

If multiple candidates started, non will win

## Raft's key idea: use random timeout before the retry

- So as to minimize the possibilities that two candidates retry at the same time
- Typically, it randomly selects a time between [T, 2T]
  - T is the election timeout

# Persistent state of each server so far

#### currentTerm

Latest term server has seen (initialized to 0 on first boot)

#### votedFor

Candidate Id that received vote in current term (or null if none)

# Basic request vote RPC so far

Invoked by candidates to gather votes.

## **Arguments:**

candidateId candidate requesting vote
term candidate's term

#### **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, grant vote and reset election timeout

# Raft's high-level approach: problem decomposition

#### 1. Leader election

Select one server as the leader

Detect crashes, choose new leader

# 2. Log replication (normal operation)

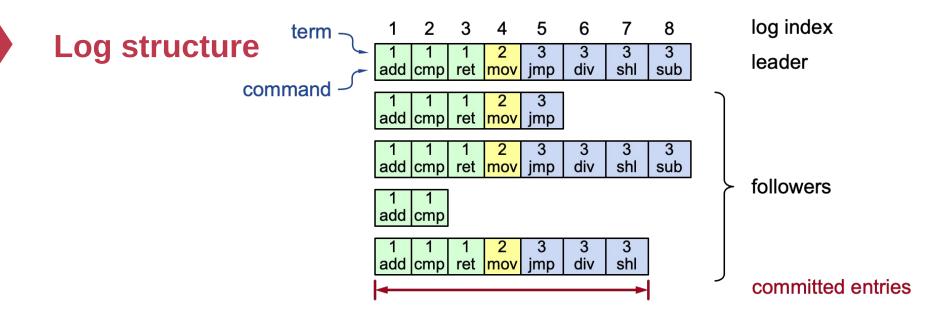
Leader accepts commands from clients, append to its log

Leader replicates its log to other servers (overwrites inconsistencies)

## 3. Safety

Keep logs consistent

Only servers with up-to-date logs can become the leader



# **Log entry = index, term, command**

Stored on the disk to tolerate failures

# A log is committed if it can be safely applied to the state machine

i.e., eventually stored on all the servers with the same value

Not all entries are committed (will talk about later)

# Persistent state of each server + log

#### currentTerm

Latest term server has seen (initialized to 0 on first boot)

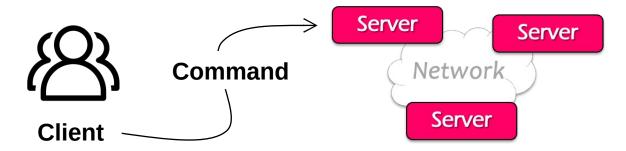
#### votedFor

Candidate Id that received vote in current term (or null if none)

# Log[]

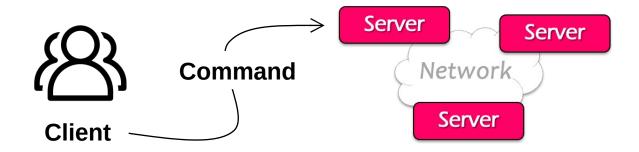
Log entries

# Normal operations to update the log



- (1) Send command to the leader
- 2 Leader appends command to its log
- 3 Leader sends AppendEntries RPCs to followers
- ④ Once a new entry (of log) committed: Leader passes command to its state machine, returns results to client Notifies followers of committed entries, Follower pass committed commands to their state machines

# Normal operations to update the log



#### Crashed/slow followers?

Leader retries RPCs until they succeed (at least once)

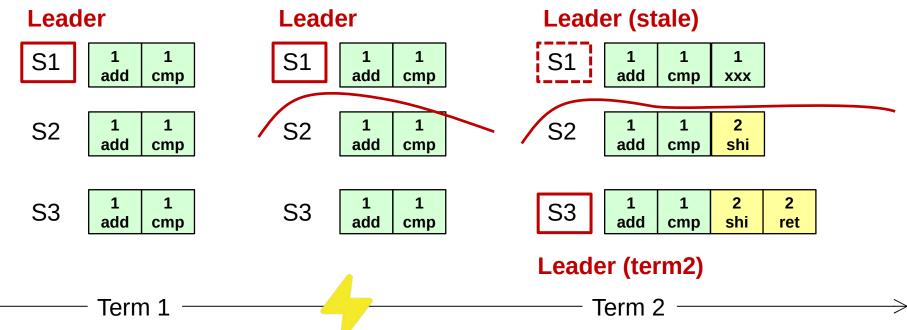
## Performance is optimal in the common case

One successful RPC to any majority of servers

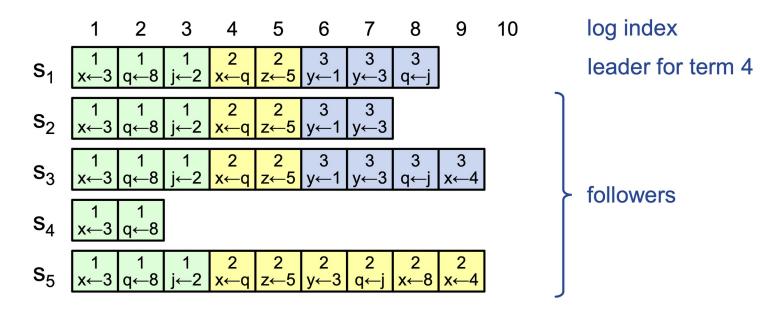
# Challenge: crash can cause log to inconsistencies

### Case: an old leader is unaware of the new leader due to network partition

Yet, it can still get commands from the clients



# Challenge: crash can cause log to inconsistencies



# Raft minimizes special code for repairing inconsistencies

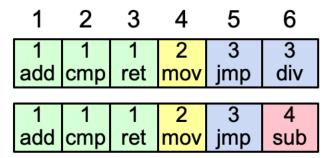
- Leaders assume its log is correct
- Normal operation will repair all inconsistencies

# **Consistency of the log**

Question: how to achieve this property?

# High level of coherency between logs maintained by the raft:

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



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

Note that not all log entries are committed

# AppendEntries consistency checks

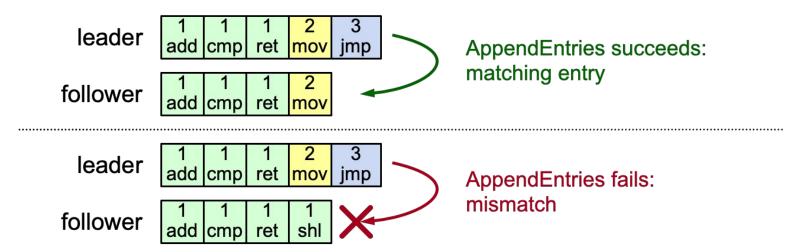
Implements an induction step, ensures coherency

## **Each RPC argument contains**

Append index, term, term of entry preceding new ones

# Follower checks whether it has the matching entry

Otherwise, it rejects the request



# Why can rejection happens? 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, w/ different position

- Leader's log is "the truth"
- Will eventually make follower's logs identical to leader's (overwrite the divergent log entries with the leader's ones)

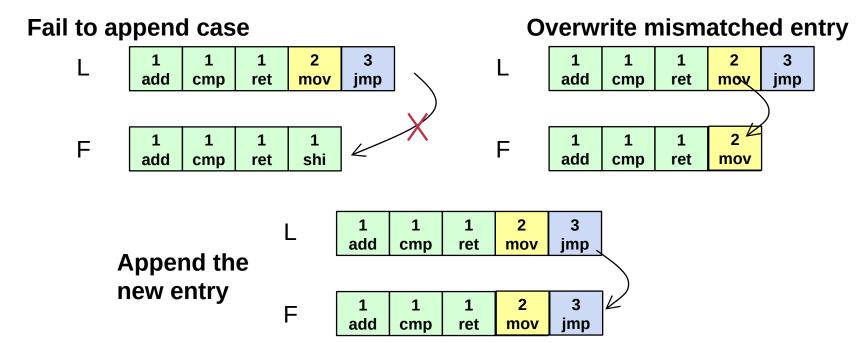
The rejected entries will be overwritten by the leader's log entry

# **Handle rejections**

Implements an induction step, ensures coherency

## In this example, the leader will

- Overwrite index 4 with [2 mov] with AppendEntries
- Then append entry [3 jmp] to the end of the follower



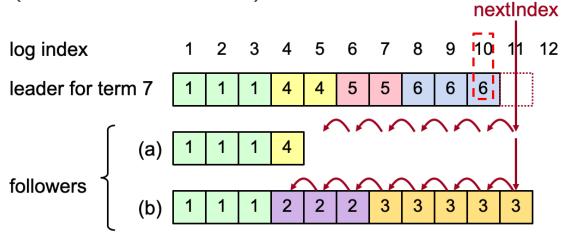
# More on Repairing Follower Logs

# 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)



# **AppendEntries RPC (Simplified)**

Invoked by leader to replicate log entries and discover inconsistencies

## **Arguments:**

**term** leader's term

**prevLogIndex** index of log entry immediately preceding new ones

prevLogTerm term of prevLogIndex entry

entries[] log entries to store (empty for heartbeat)

#### **Results:**

**success** true if follower contained entry matching prevLogIndex and prevLogTerm

# **AppendEntries RPC (Simplified)**

## **Implementation:**

- 1.Return false 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

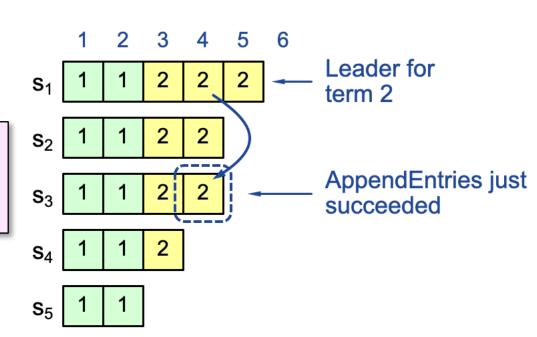
# Overwrite makes a subtle issue: When can we commit a log entry? (Since an appended log entry may be overwritten)

# Intuitive result: replicating on a majority

If the log is replicated on a majority of servers, it can be replicated on the state machine

Not always true on raft so far

Question: can index 4 be overwritten after appending to S3?

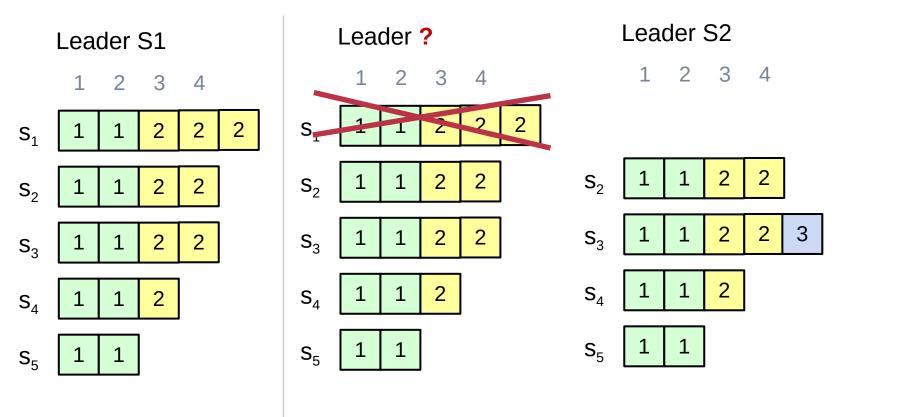


# **Case study: raft overwriting**

Term 2

Question: can index 4 be overwritten?

Term 3

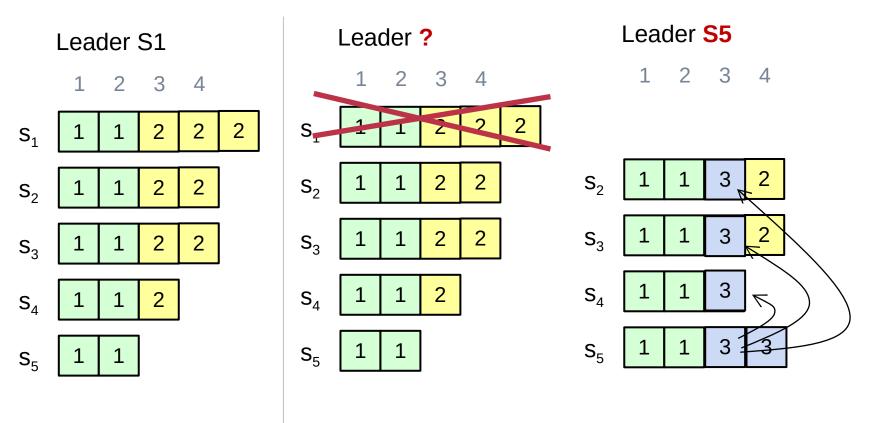


S1 crashes

# **Case study: raft overwriting**

Term 2

Question: can index 4 be overwritten?



S1 crashes

Term 3

# Safety requirement of the commit entry

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 (no overwritten)

#### This guarantees the safety requirement

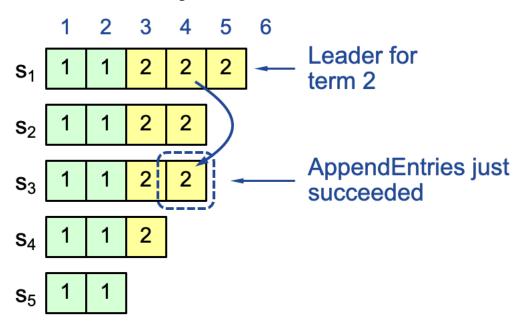
- Leaders never overwrite entries in their logs
- Only entries in the leader's log can be committed
- Entries must be committed before applying to state machine



# Goal: if a log entry has been replicated to majority followers, it is <u>likely</u> to commit

# **Committing Entry from the Current Term**

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



Question: how to prevent index 4 from being overwritten? Prevent S4 or S5 from becoming the leader

# **Picking the Best Leader**

# During elections, choose candidate with log most likely to contain all committed entries

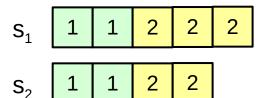
- Candidates include log info in RequestVote RPCs (index & term of last log entry)
- Voting server V denies vote if its log is "more complete":

```
(lastTermV > lastTermC) ||
(lastTermV == lastTermC) && (lastIndexV > lastIndexC)
```

Leader will have "most complete" log among electing majority

# **Comparing the best leader**

```
(lastTermV > lastTermC) ||
(lastTermV == lastTermC) && (lastIndexV > lastIndexC)
```

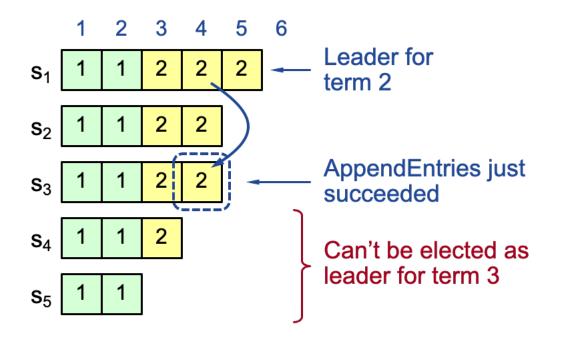


Question: which log is longer?

S1 & S2, respectively

### **Committing Entry from the Current Term**

Leader decides entry in current term is committed



Safe: leader for term 3 must contain entry 4

# Complete picture of request vote 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

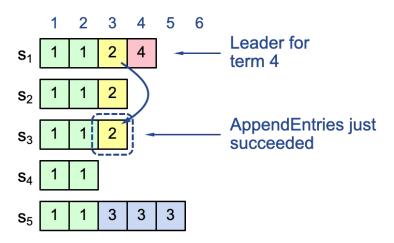
#### Commit rule for raft so far

#### If the log entry from the leader term is replicated to a majority of followers

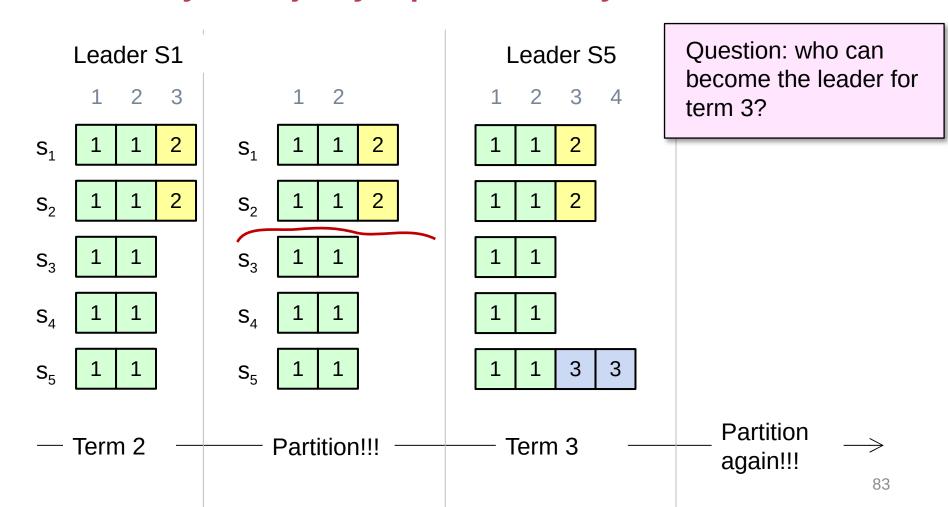
- Then we can treat it as committed
  - The later leader must contain the entry

#### But, what about replicating log entry from a previous term?

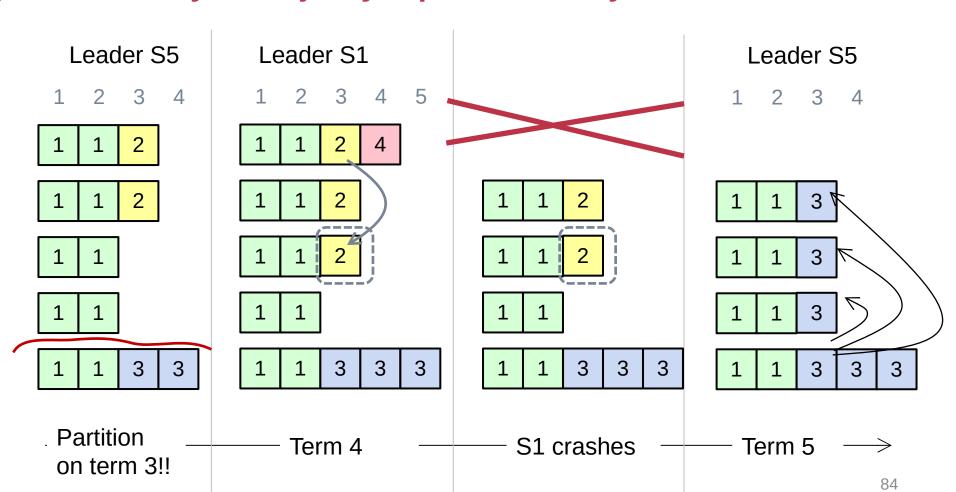
The previous term's entry may fail to reach a majority



# Case study: a majority replicated entry can be overwritten

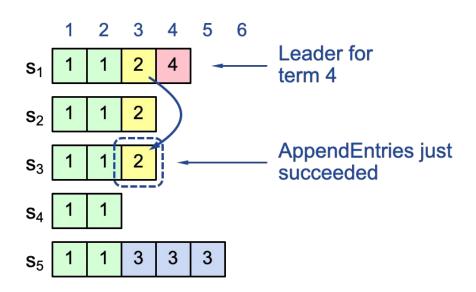


# Case study: a majority replicated entry can be overwritten



# **Committing Entry from Earlier Term**

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



#### **Entry 3 not safely committed:**

- S5 can still be elected as leader for term 5
- If elected, it will overwrite entry 3 on S1, S2 and S3 (recall our previous example)

#### **Commit rules**

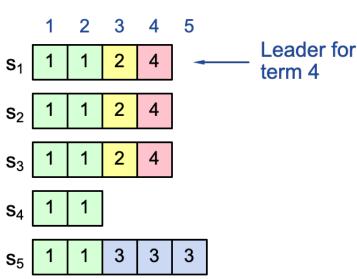
#### For a leader to decide an (previous) entry is committed:

- 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

#### This is because once entry 4 committed:

- s<sub>5</sub> cannot be elected leader for term 5
- Entries 3 and 4 both safe

Combination of election rules & commitment rules makes Raft safe



# **Neutralizing Old Leaders**

#### Deposed leader may not be dead:

- 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 the 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