RAFT:
In Search of an Understandable Consensus Algorithm

Diego Ongaro
John Ousterhout

USENIX Annual Technical Conference 2014

Slides from a talk by Ongaro and Ousterhout, and a talk by Ion Stoica.
This version edited by S. Sudarshan, IIT Bombay

Motivation (I)

"Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members."

- Very important role in building fault-tolerant distributed systems

Motivation (II)

<u>Paxos</u>

- Current standard for both teaching and implementing consensus algorithms
- Very difficult to ***understand*** and very hard to ***implement***

<u>Raft</u>

- New protocol (2014)
- Much easier to ***understand***
- Several ***open-source implementations***

Paxos Limitations

"The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-)." – NSDI reviewer

"There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system…the final system will be based on an unproven protocol." – Chubby authors

Replicated state machines

Allows a collection of servers to

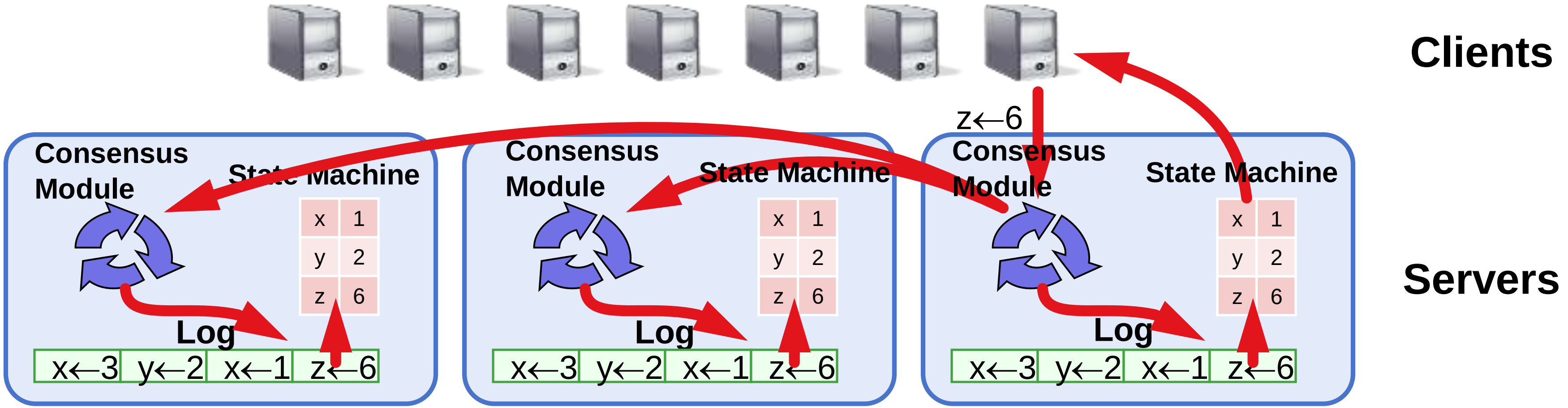
- Maintain identical copies of the same data
- Continue operating when some servers are down
  - A majority of the servers must remain up

Many applications

Typically built around a distributed log

Each server stores a log containing commands

Replicated State Machines



Replicated log ⇒ replicated state machine

- All servers execute same commands in same order

Consensus module ensures proper log replication

The distributed log (I)

State machines always execute commands
in the log order

- They will remain consistent as long as command executions have ***deterministic results***

System makes progress as long as any majority of servers are up

Failure model: fail-stop (not Byzantine), delayed/lost messages

Designing for understandability

Main objective of RAFT

- Whenever possible, select the alternative that is the easiest to understand

Techniques that were used include

- Dividing problems into smaller problems
- Reducing the number of system states to consider

Raft Overview

1. Leader election
   - Select one of the servers to act as cluster leader
   - Detect crashes, choose new leader

2. Log replication (normal operation)
   - Leader takes commands from clients, appends them to its log
   - Leader replicates its log to other servers (overwriting inconsistencies)

3. Safety
   - Only a server with an up-to-date log can become leader

Raft basics: the servers

A RAFT cluster consists of several servers

- Typically five

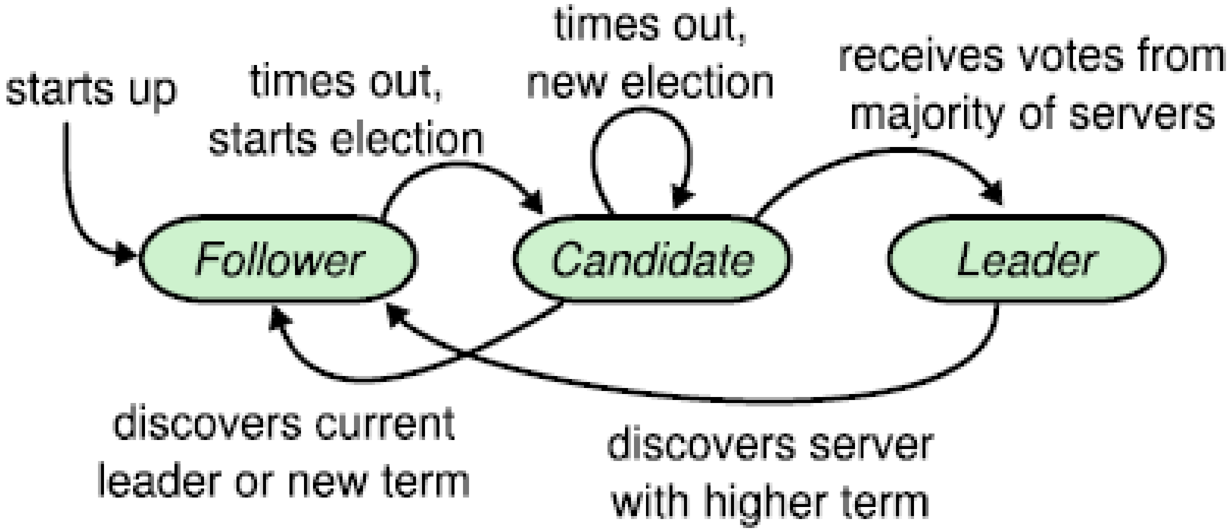
Each server can be in one of three states

- ***Leader***
- ***Follower***
- ***Candidate*** (to be the new leader)

Followers are passive:

- Simply reply to requests coming from their leader

Server states



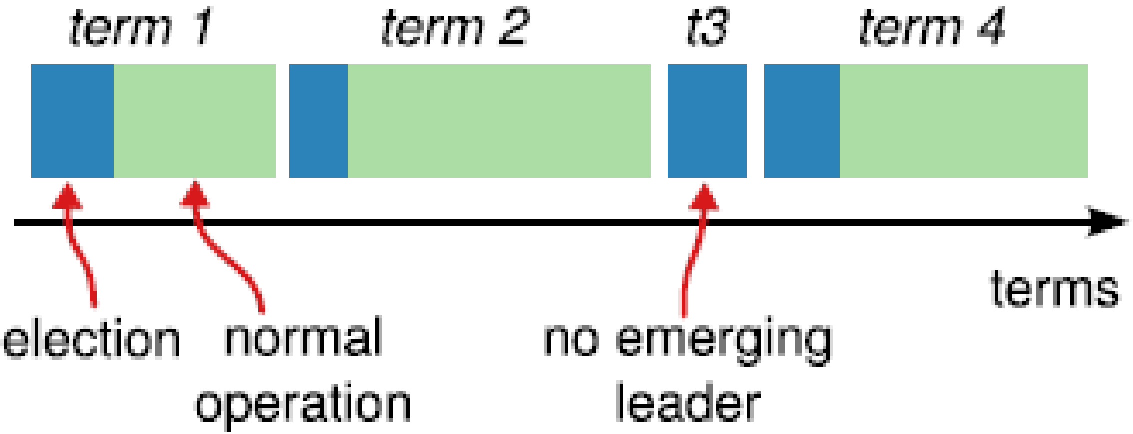
Raft basics: terms (I)

Epochs of arbitrary length

- Start with the election of a leader
- End when
  - Leader becomes unavailable
  - No leader can be selected (split vote)

Different servers may observe transitions between terms at different times or even miss them

Raft basics: terms (II)



Raft basics: RPC

Servers communicate though idempotent RPCs

RequestVote

- Initiated by candidates during elections

AppendEntry: Initiated by leaders to

- Replicate log entries
- Provide a form of heartbeat
  - Empty AppendEntry( ) calls

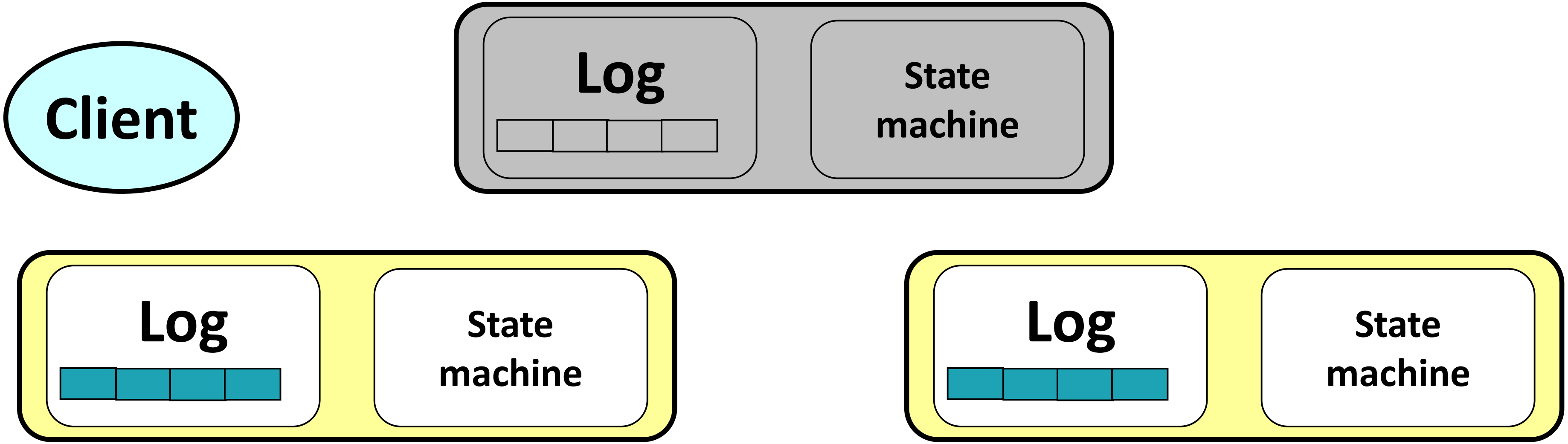
Leader elections

Servers start being ***followers***

Remain followers as long as they receive valid RPCs from a leader or candidate

When a follower receives no communication over a period of time (the ***election timeout***), it starts an election to pick a ***new leader***

The leader fails

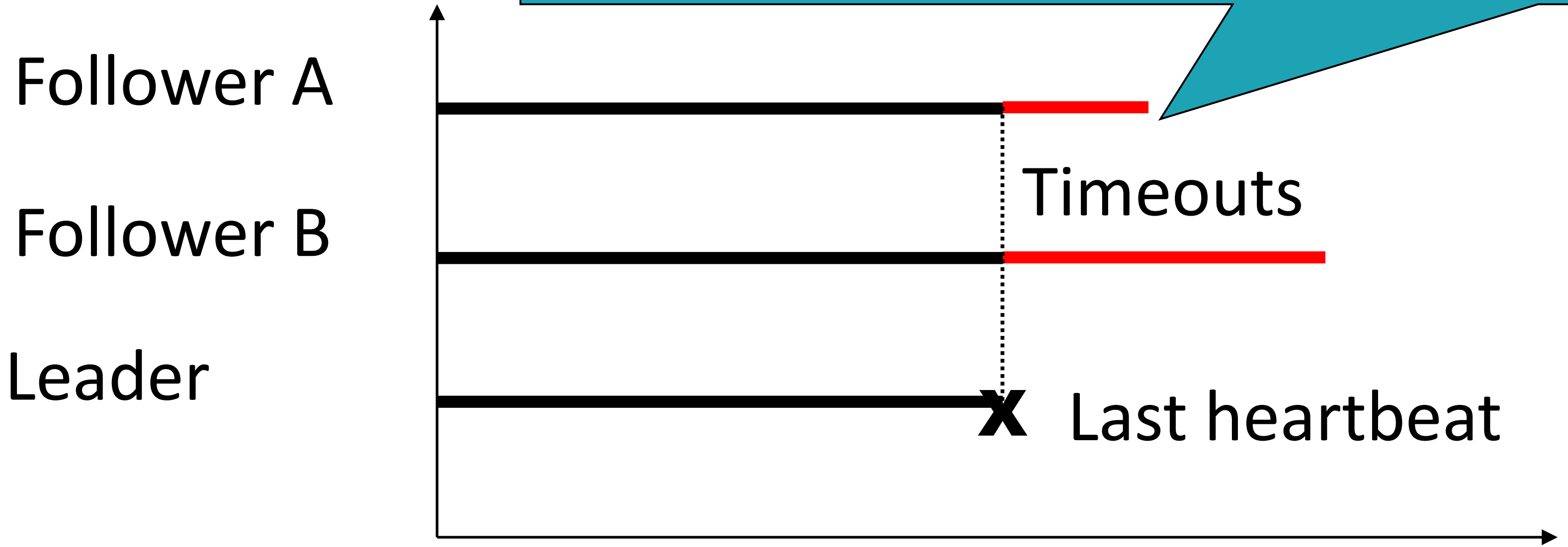


Followers notice at ***different times*** the lack of heartbeats

Decide to elect a new leader

Example

Follower with the ***shortest timeout*** requests to become the ***new leader***



Starting an election

When a follower starts an election, it

- Increments its current term
- Transitions to candidate state
- Votes for itself
- Issues ***RequestVote*** RPCs in parallel to all the other servers in the cluster.

Acting as a candidate

A candidate remains in that state until

- It wins the election
- Another server becomes the new leader
- A period of time goes by with no winner

Winning an election

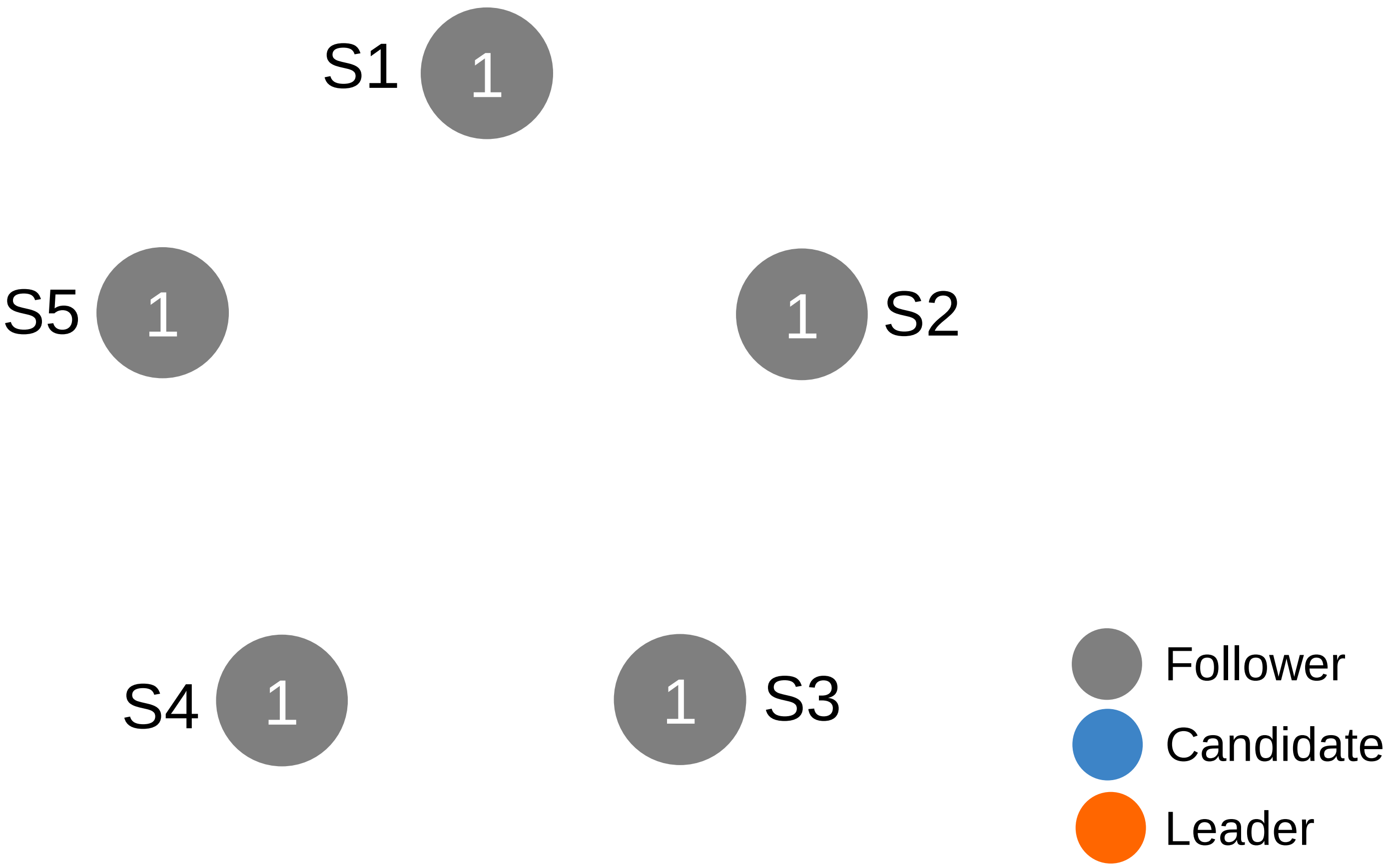
Must receive votes from a majority of the servers in the cluster for the same term

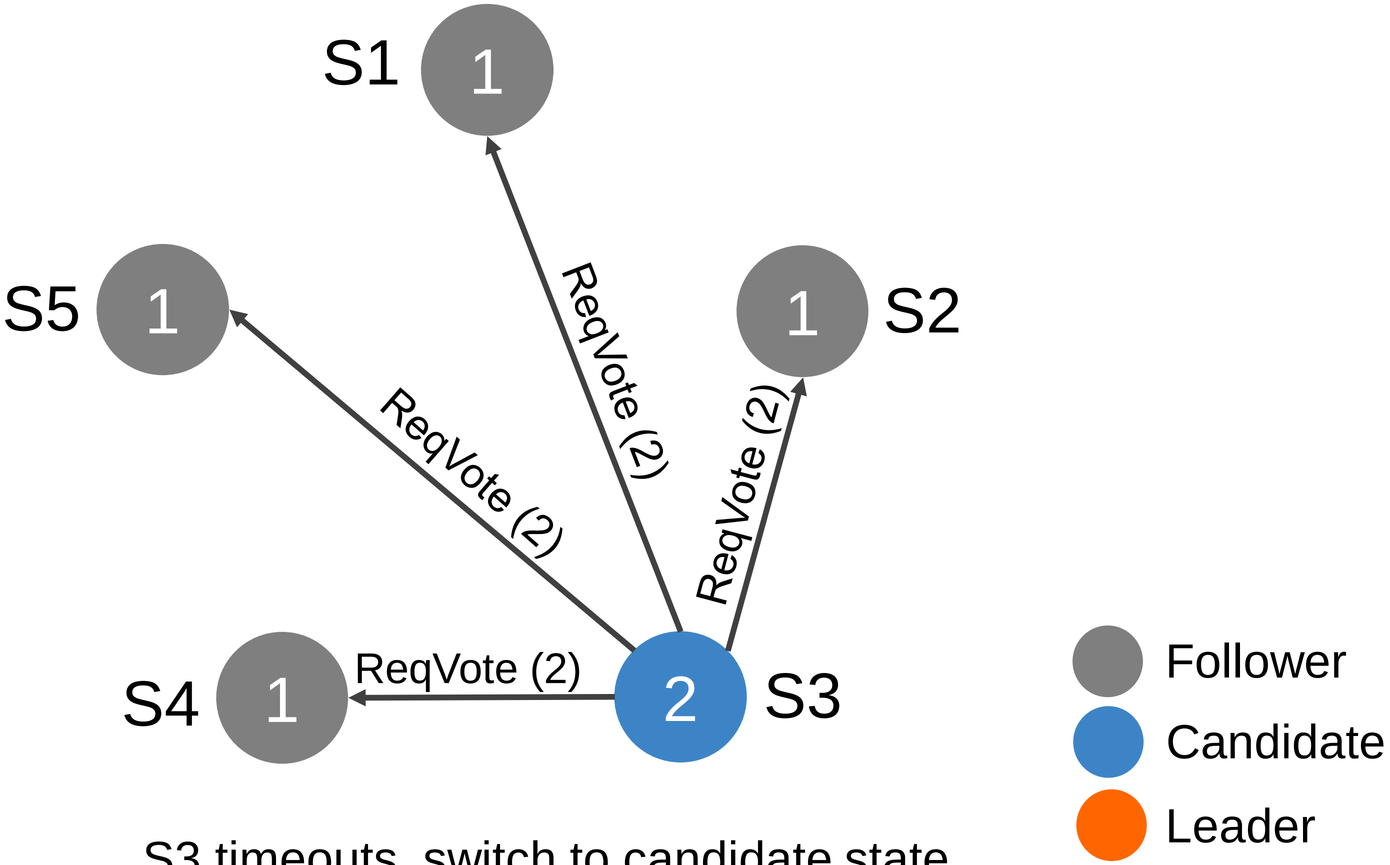
- Each server will vote for at most one candidate in a given term
  - The first one that contacted it

Majority rule ensures that at most one candidate can win the election

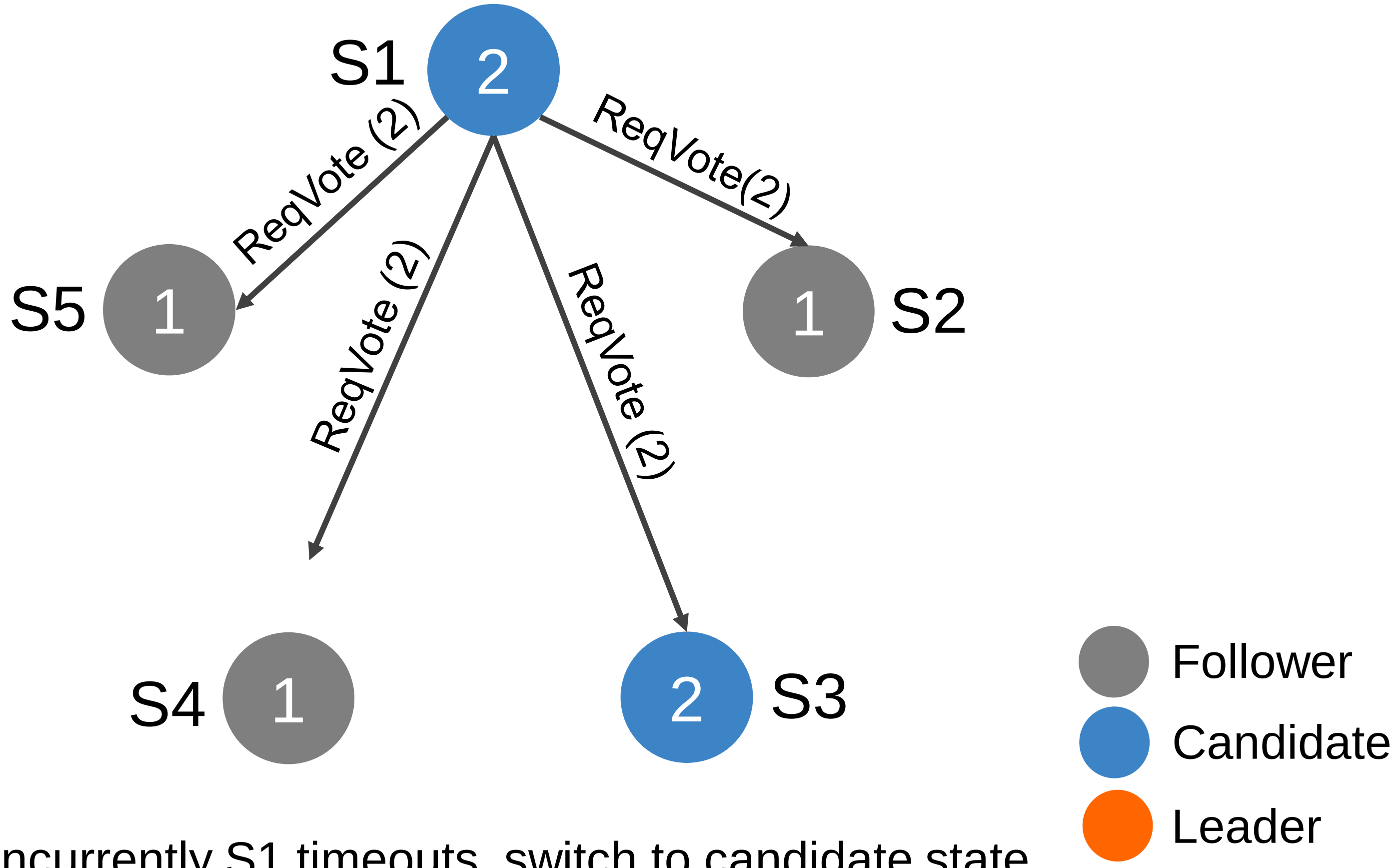
Winner becomes ***leader*** and sends heartbeat messages to all of the other servers

- To assert its new role

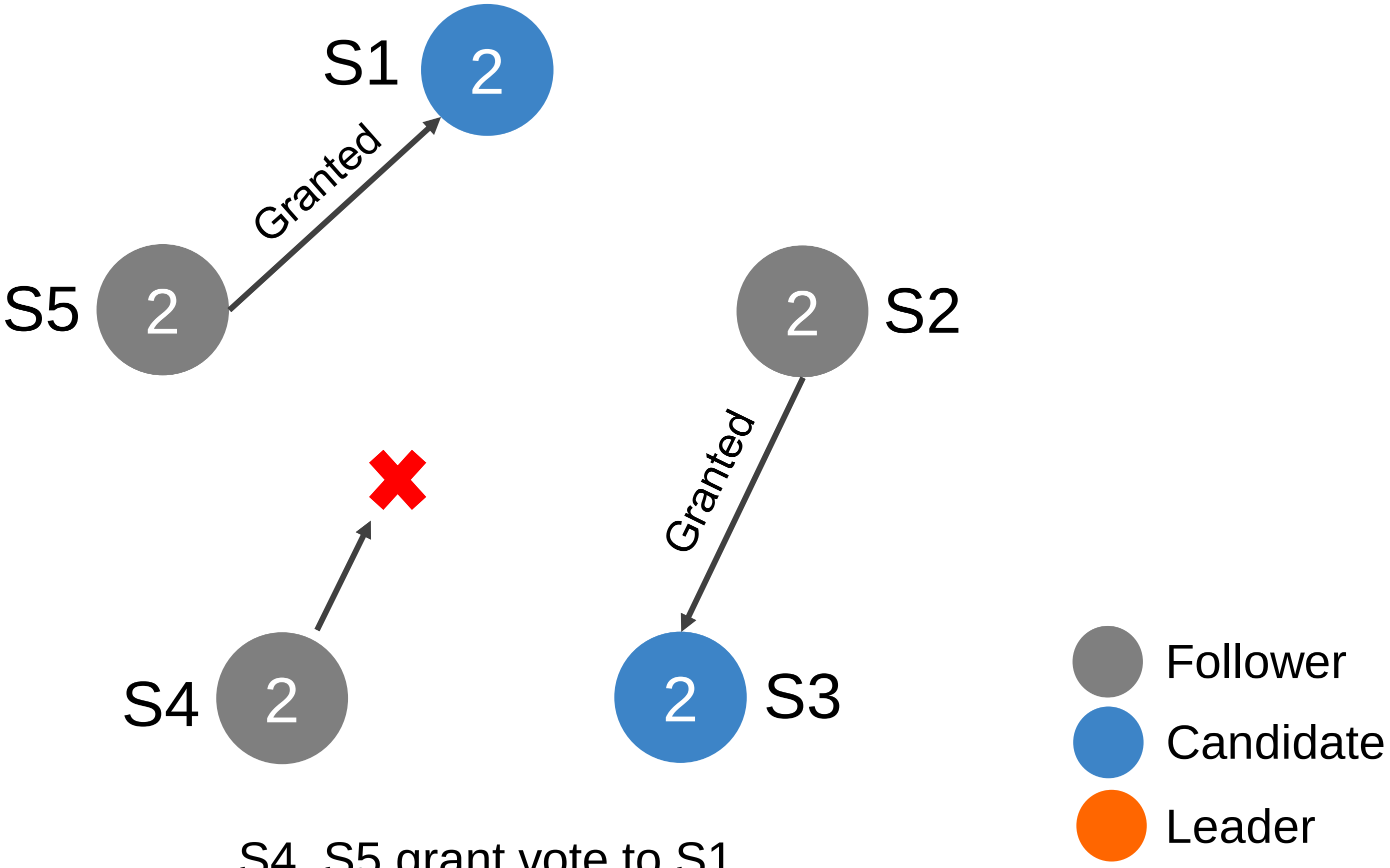

S1
1
S5
1
1
S2
S4
1
1
S3
Follower
Candidate
Leader



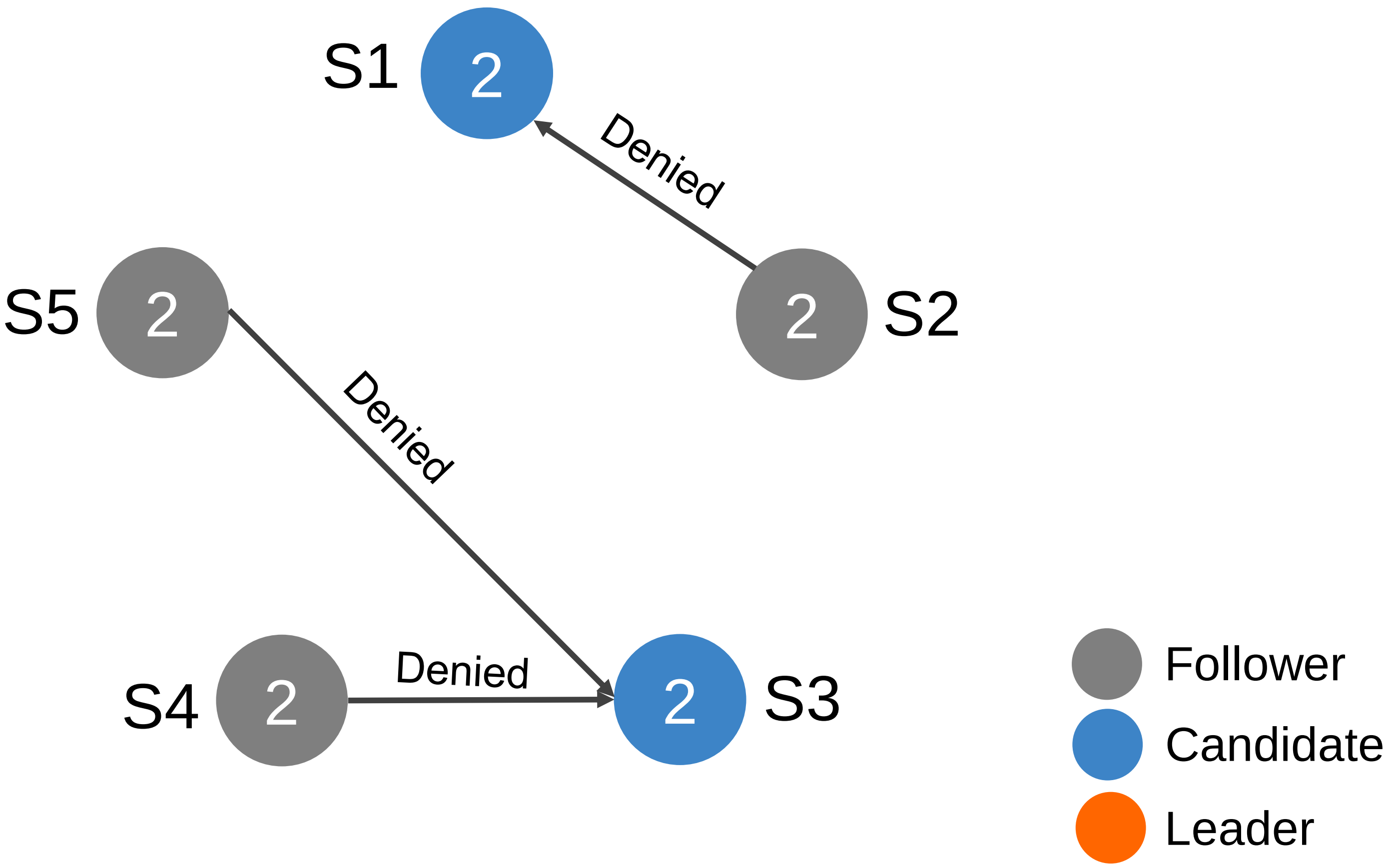
S3 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm



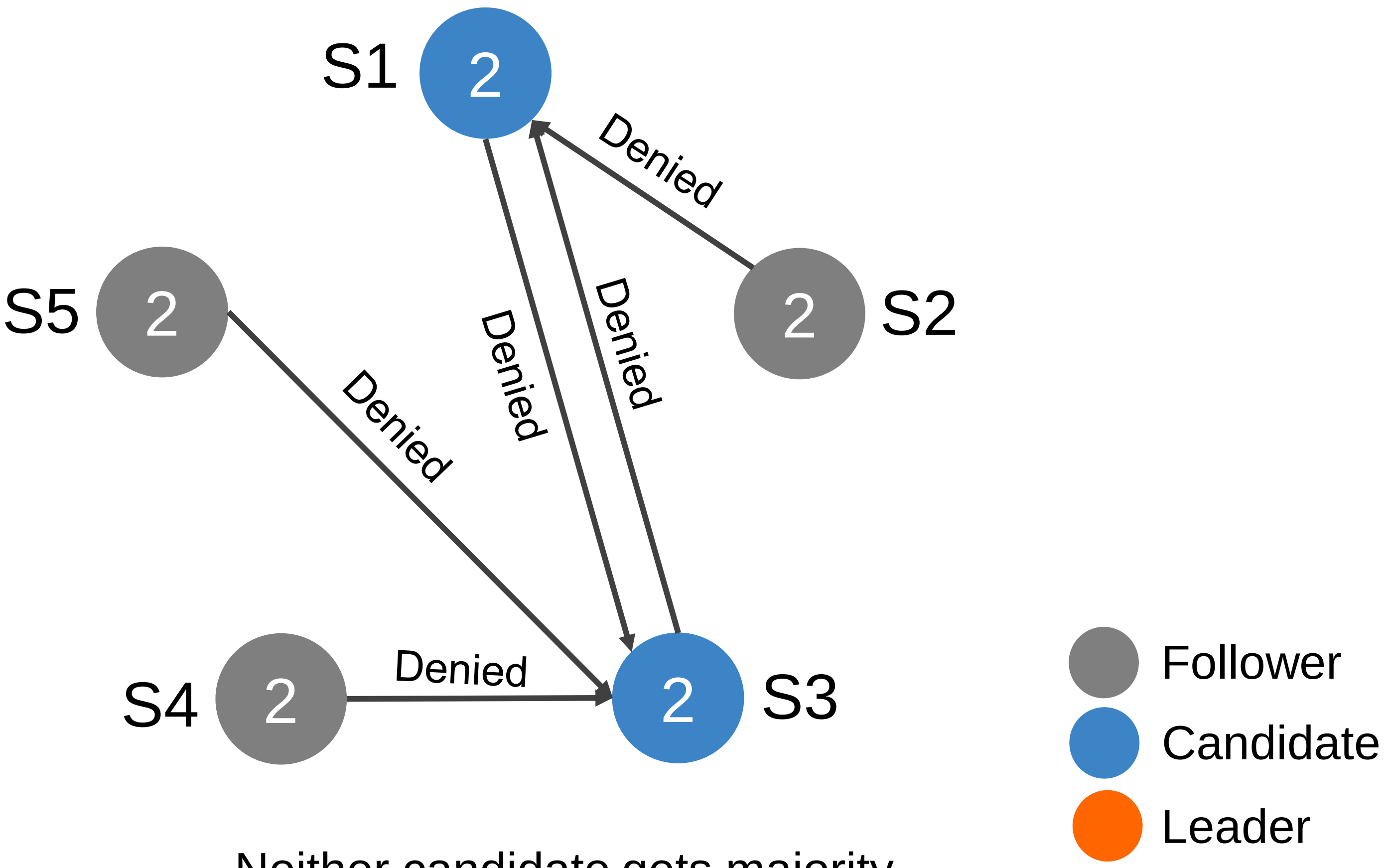
Concurrently S1 timeouts, switch to candidate state,
increment term, vote itself as a leader and ask everyone else to confirm



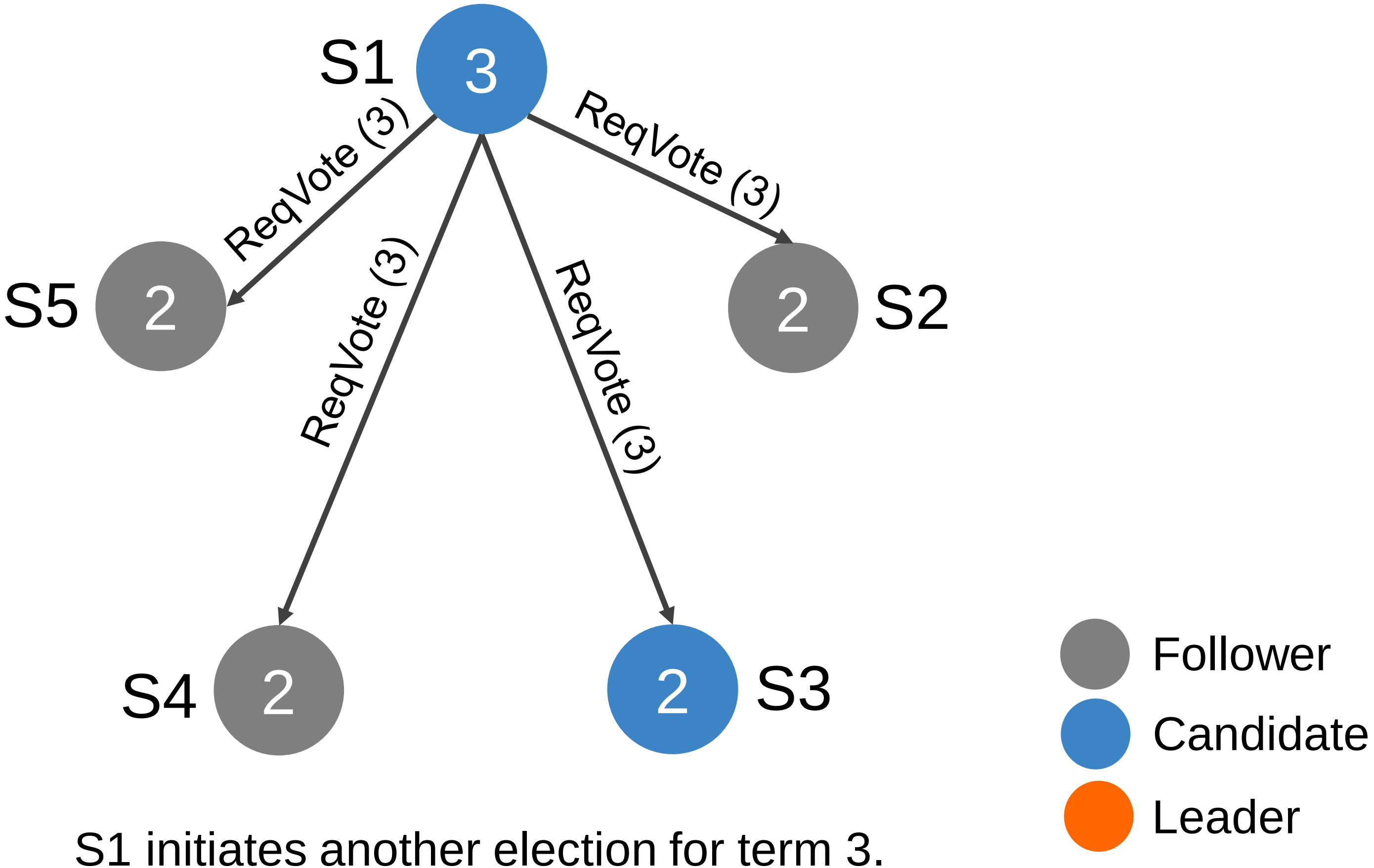
S4, S5 grant vote to S1
S2 grants vote to S3



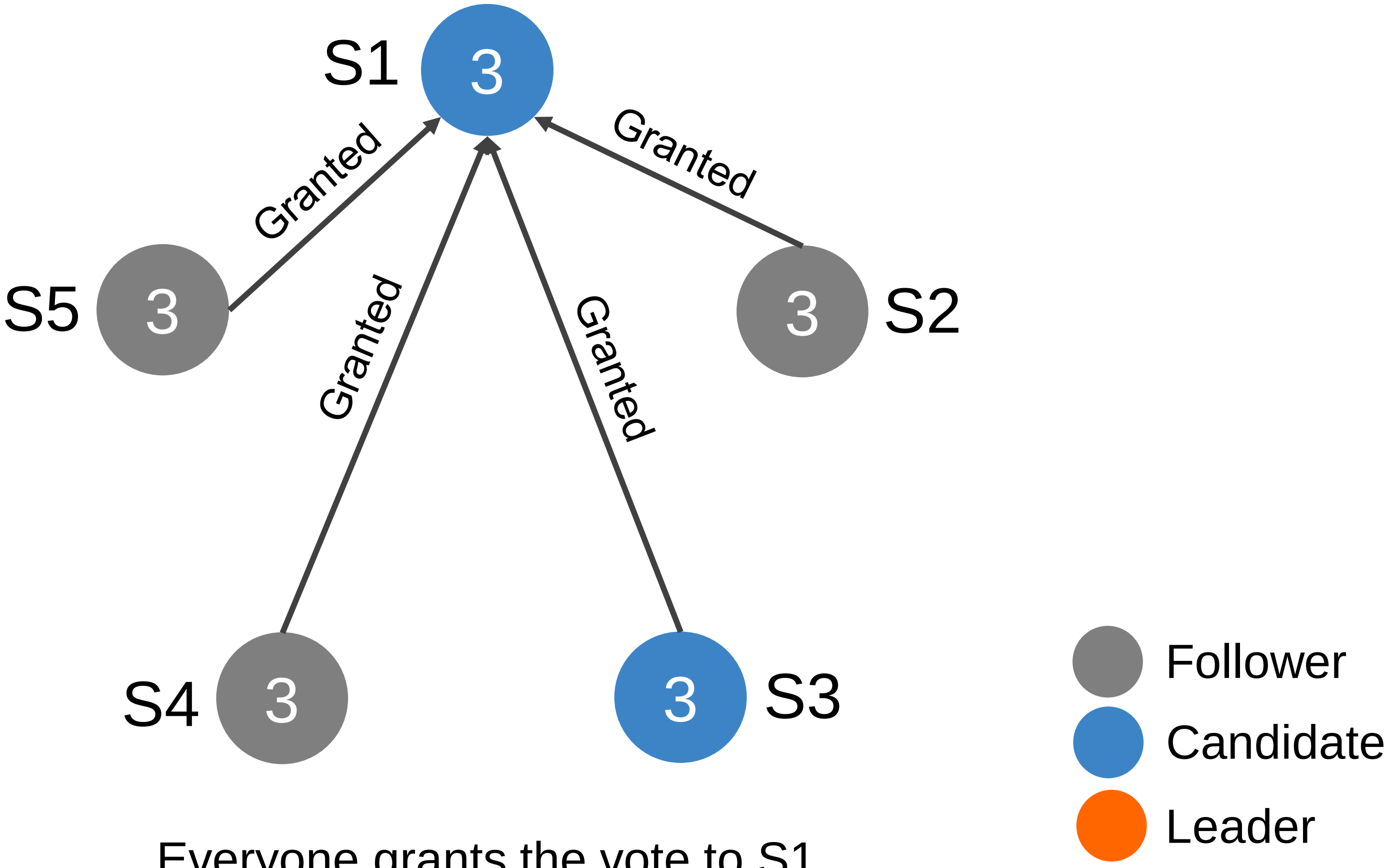
S1
2
Denied
S5
2
2
S2
Denied
S4
2
Denied
2
S3
Follower
Candidate
Leader



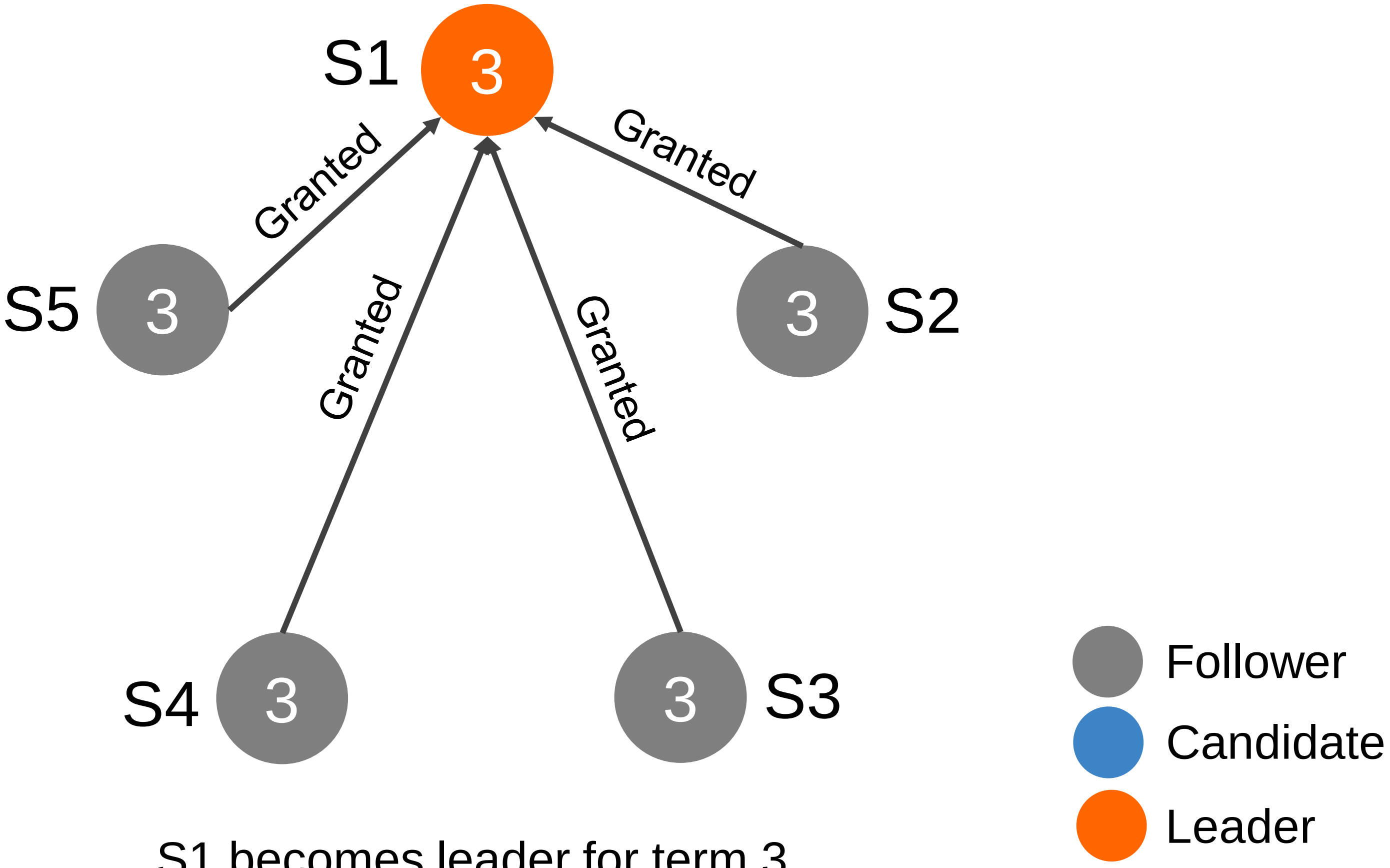
Neither candidate gets majority.
After a random delay between 150-300ms try again.



S1 initiates another election for term 3.



Everyone grants the vote to S1



S1 becomes leader for term 3,
and the others become followers.

Hearing from other servers

Candidates may receive an ***AppendEntries*** RPC from another server claiming to be leader

If the leader's term is at greater than or equal to the candidate's current term, the candidate recognizes that leader and returns to follower state

Otherwise the candidate ignores the RPC and remains a candidate

Split elections

No candidate obtains a majority of the votes in the servers in the cluster

Each candidate will time out and start a new election

- After incrementing its term number

Avoiding split elections

Raft uses randomized election timeouts

- Chosen randomly from a fixed interval

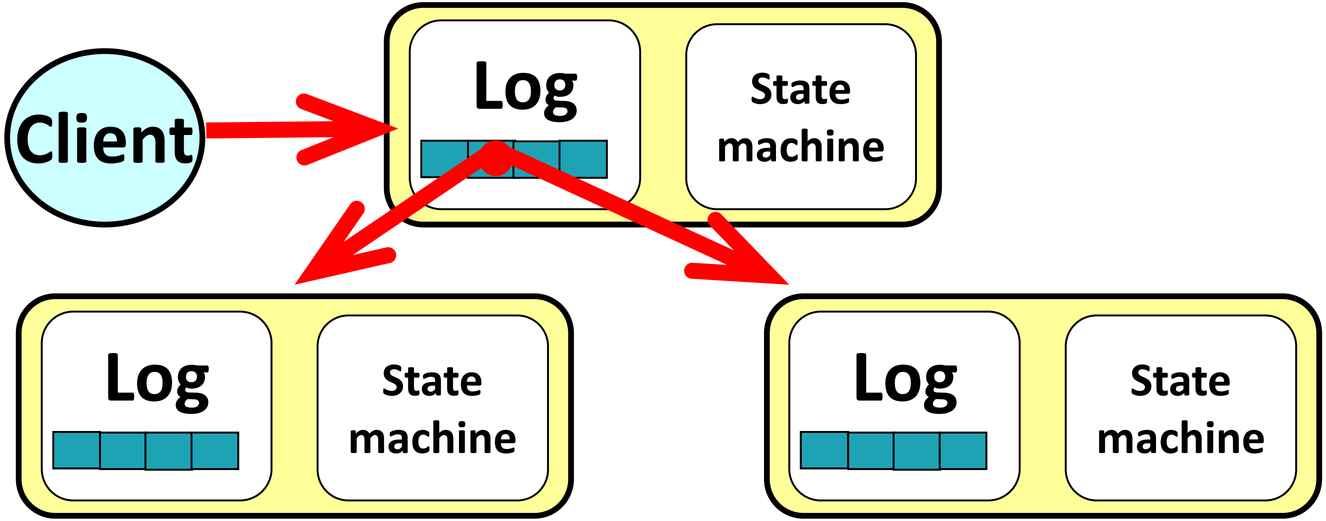
Increases the chances that a single follower will detect the loss of the leader before the others

Log replication

Leaders

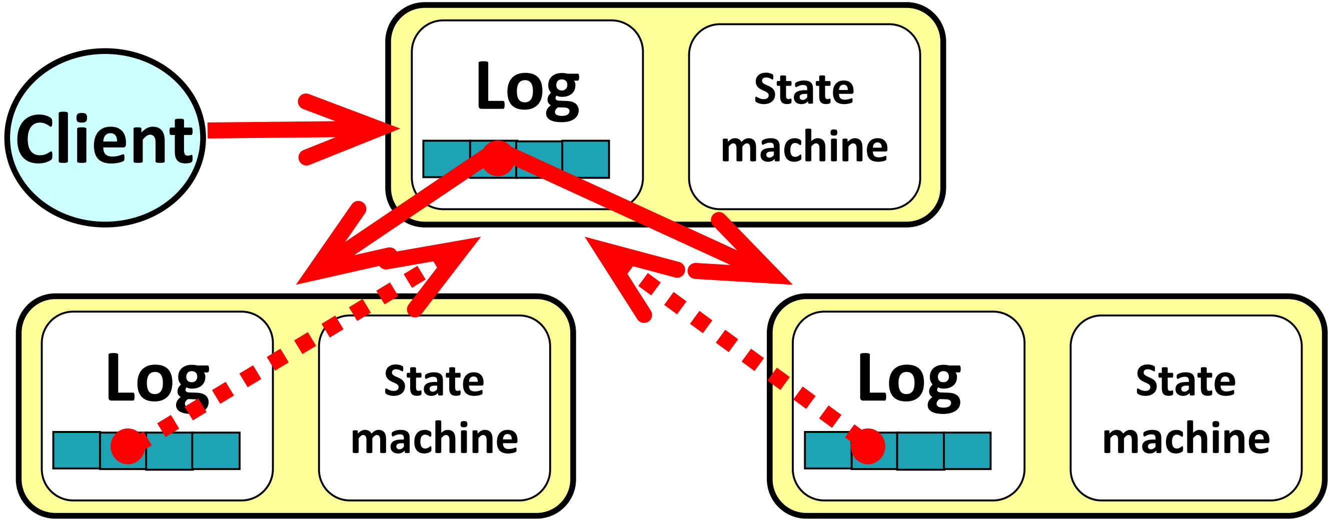
- Accept client commands
- Append them to their log (new entry)
- Issue **AppendEntry** RPCs in parallel to all followers
- Apply the entry to their state machine once it has been safely replicated
  - Entry is then ***committed***

A client sends a request



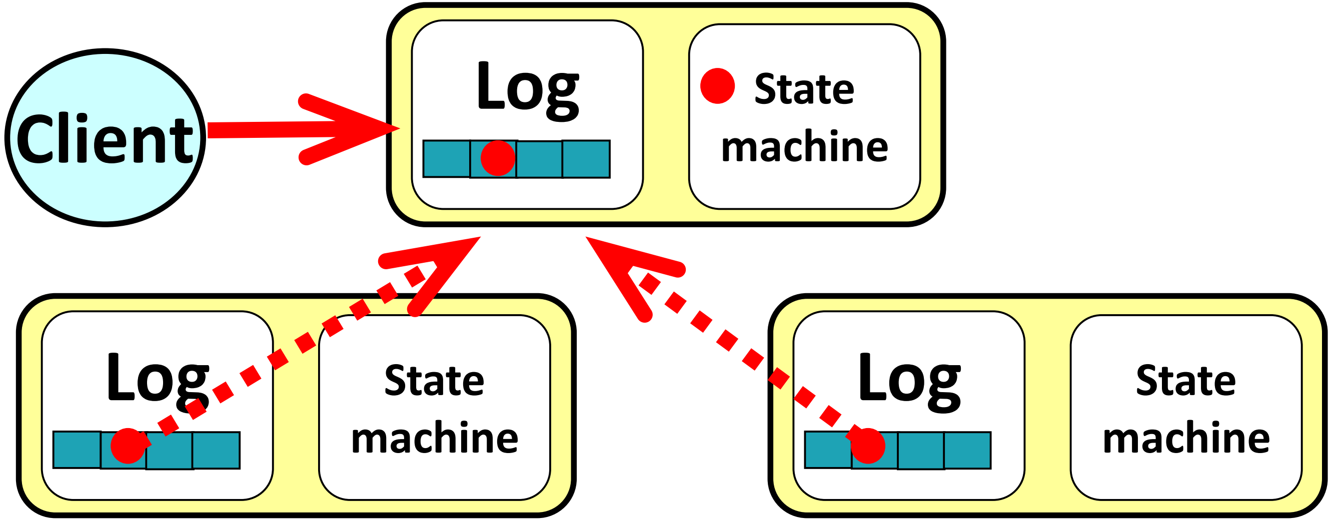
Leader stores request on its log and forwards it to its followers

The followers receive the request



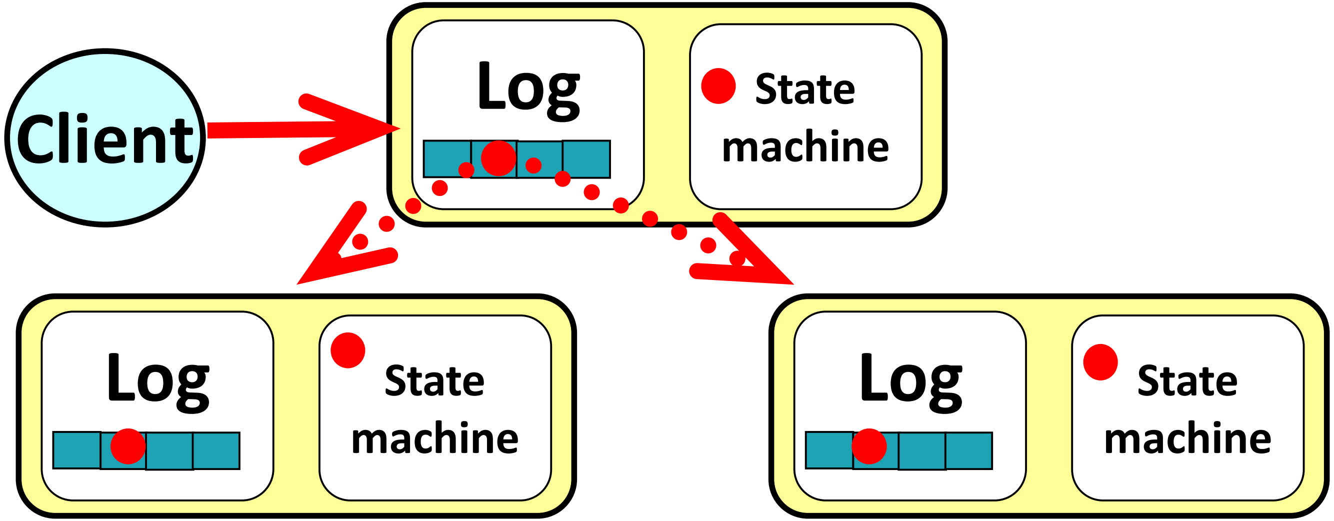
Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



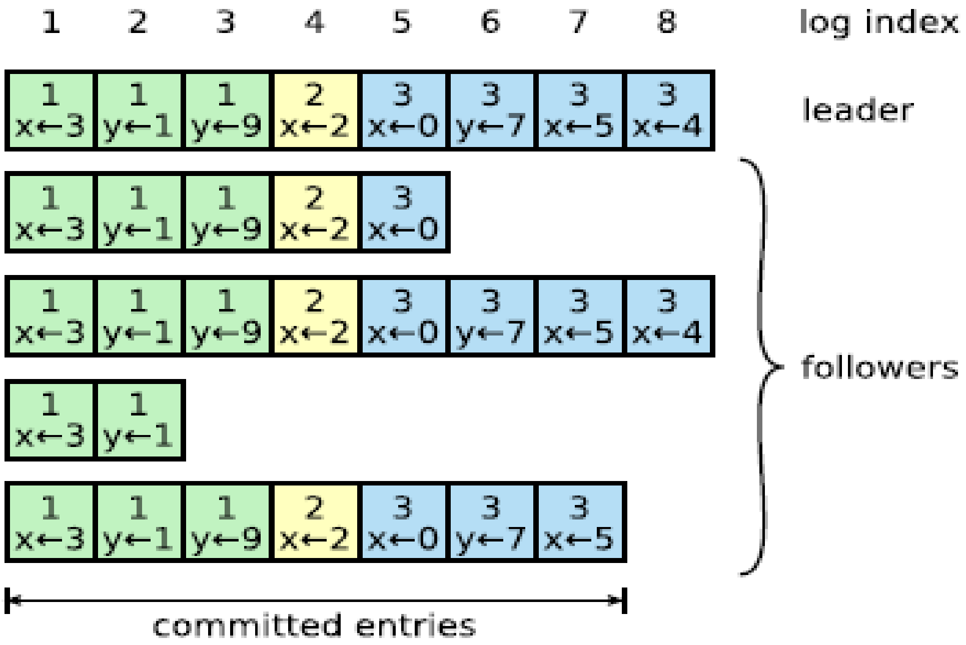
Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



Leader's heartbeats convey the news to its followers: they update their state machines

Log organization

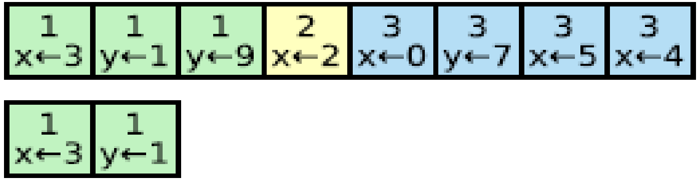


Colors identify terms

Raft log matching property

If two entries in different logs have the same index and term

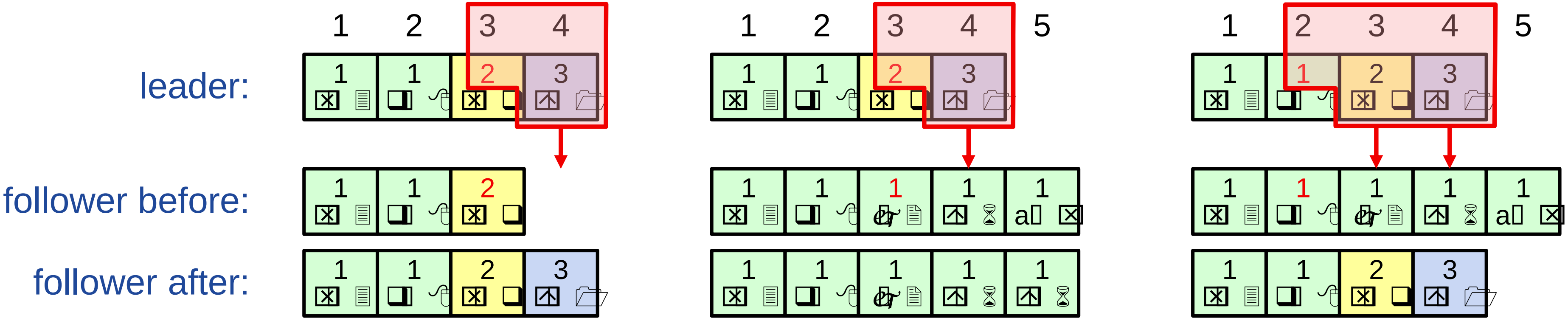
- These entries store the same command
- ***All previous entries*** in the two logs are ***identical***



AppendEntries Consistency Check

AppendEntries RPCs include <index, term> of entry preceding new one(s)

- Follower must contain matching entry; otherwise it rejects request
    - Leader retries with lower log index
- Implements an induction step, ensures Log Matching Property



Example #1: success   Example #2: mismatch   Example #3: success

Why?

Raft commits entries in ***strictly sequential order***

- Requires followers to accept log entry appends in the same sequential order
  - ***Cannot "skip" entries***

Greatly simplifies the protocol

Handling slow followers ,…

Leader reissues the AppendEntry RPC

- They are idempotent

Committed entries

Guaranteed to be both

- Durable
- Eventually executed by all the available state machine

Committing an entry also commits all previous entries

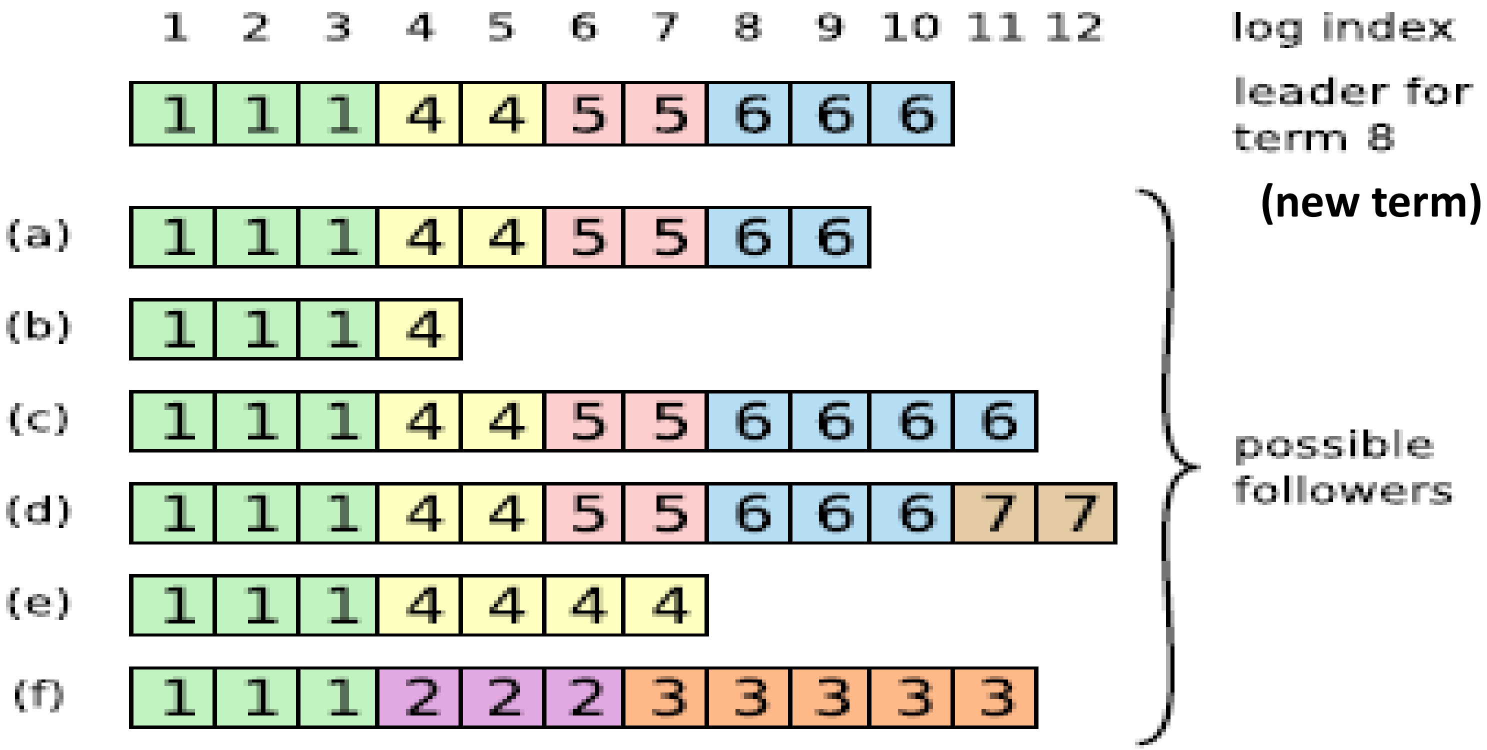
- All AppendEntry RPCs—including heartbeats—include the index of its most recently committed entry

Handling leader crashes (I)

Can leave the cluster in a inconsistent state if the old leader had not fully replicated a previous entry

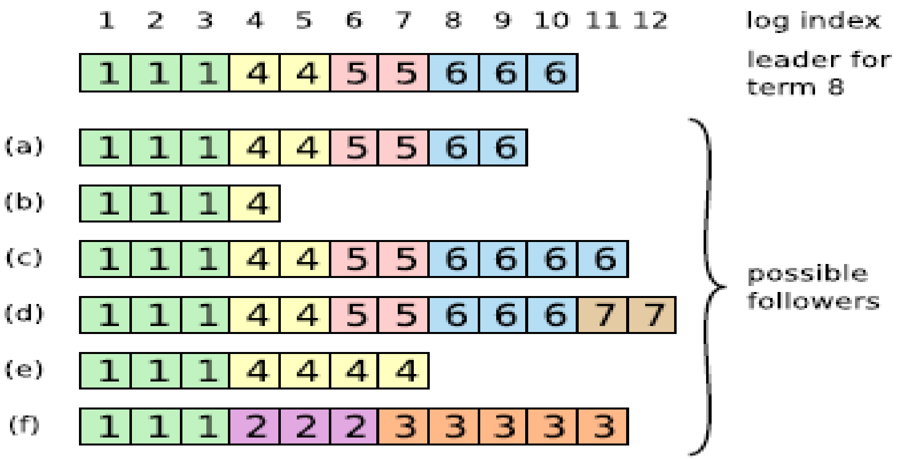
- Some followers may have in their logs entries that the new leader does not have
- Other followers may miss entries that the new leader has

Handling leader crashes (II)

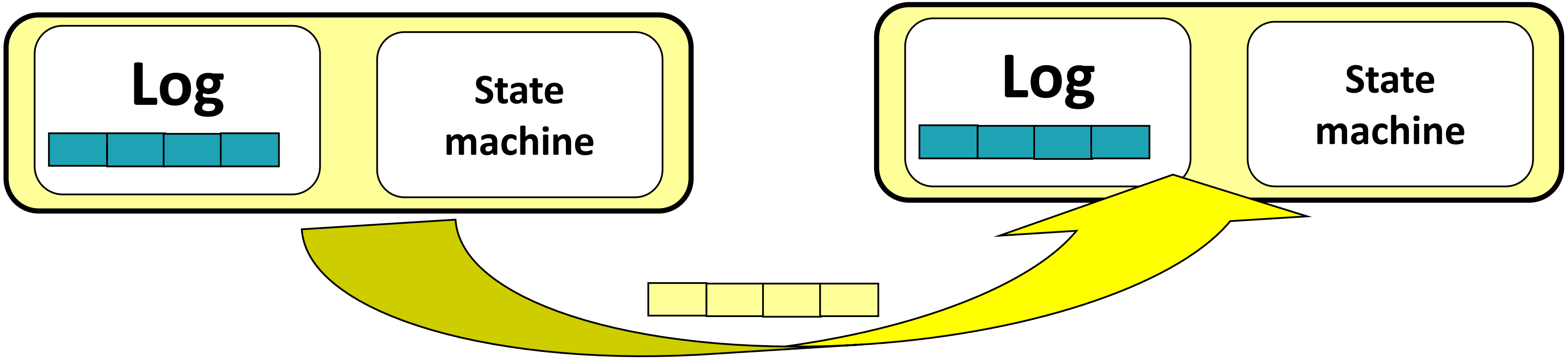


Log Status After Election

- When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs.
- Each box represents one log entry; the number in the box is its term.
- A follower may be missing entries (a–b),
- May have extra uncommitted entries (c–d), or both (e–f). or several terms.
  - E.g. scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.



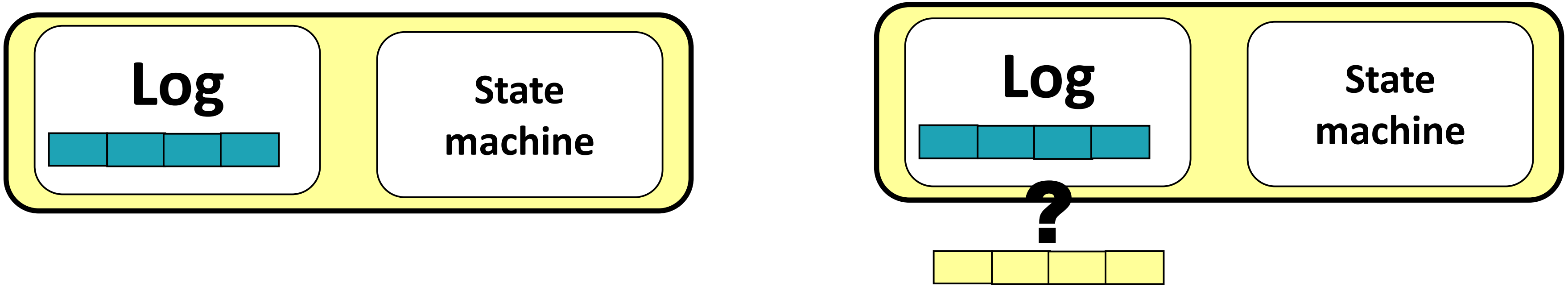
An election starts



Candidate for leader position requests votes of other former followers

- Includes a summary of the state of its log

Former followers reply



Former followers compare the state of their logs with credentials of candidate

Vote for candidate unless

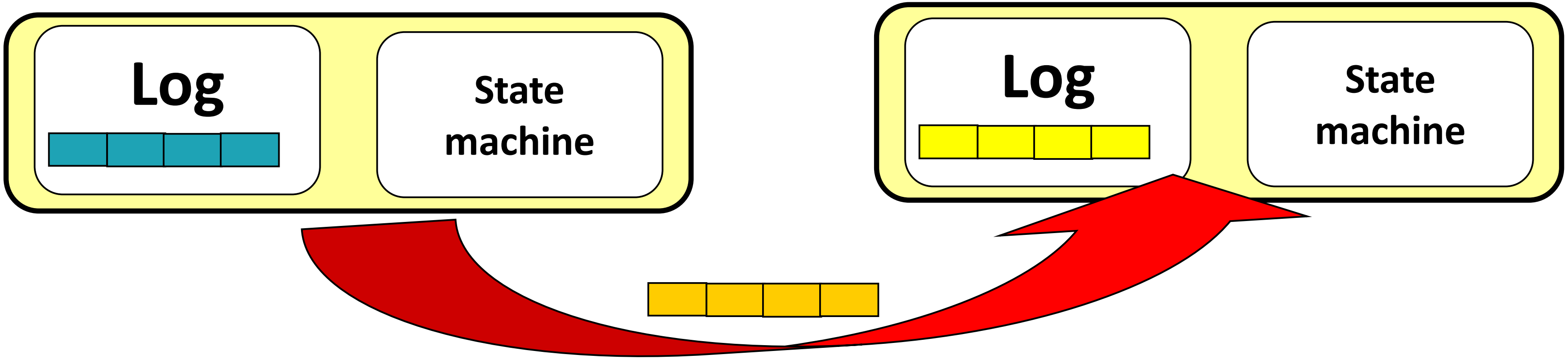
- Their own log is more "up to date"
- They have already voted for another server

Handling leader crashes (III)

Raft solution is to let the new leader to force followers' log to duplicate its own

- Conflicting entries in followers' logs will be ***overwritten***

The new leader is in charge



Newly elected candidate forces all its followers to duplicate in their logs the contents of its own log

How? (I)

Leader maintains a ***nextIndex*** for each follower

- Index of entry it will send to that follower

New leader sets its ***nextIndex*** to the index ***just after its last log entry***

- 11 in the example

Broadcasts it to all its followers

How? (II)

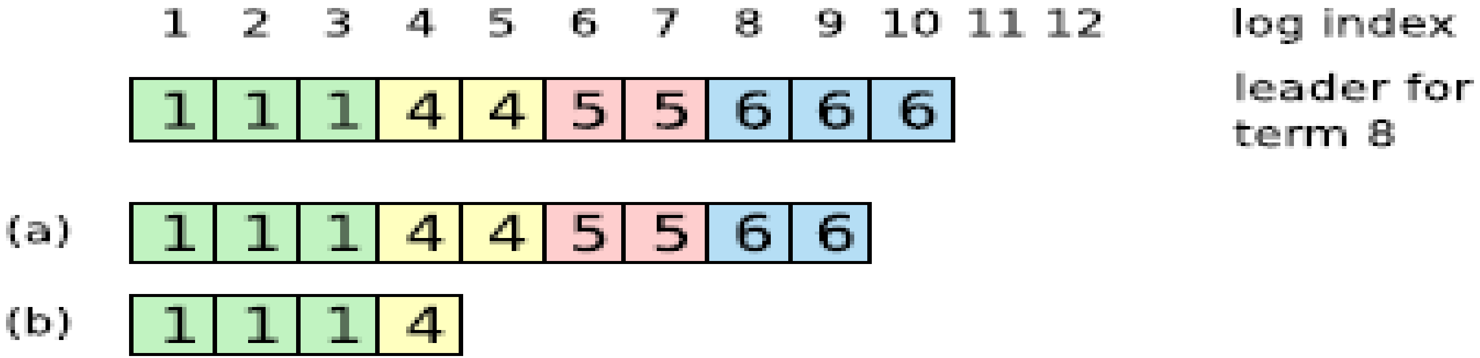
Followers that have missed some AppendEntry calls will refuse all further AppendEntry calls

Leader will decrement its nextIndex for that follower and redo the previous AppendEntry call

- Process will be repeated until a point where the logs of the leader and the follower **match**

Will then send to the follower all the log entries it missed

How? (III)



By successive trials and errors, leader finds out that the first log entry that follower (b) will accept is log entry 5

It then forwards to (b) log entries 5 to 10

Interesting question

How will the leader know which log entries it can commit

- Cannot always gather a majority since some of the replies were sent to the old leader

Fortunately for us, any follower accepting an AcceptEntry RPC implicitly acknowledges it has processed all previous AcceptEntry RPCs

Followers' logs cannot skip entries

A last observation

Handling log inconsistencies does not require a special sub algorithm

- Rolling back EntryAppend calls is enough

Safety

Two main questions

1. What if the log of a new leader did not contain all previously committed entries?
   - Must impose conditions on new leaders

2. How to commit entries from a previous term?
   - Must tune the commit mechanism

Election restriction (I)

The log of any new leader ***must*** contain all previously committed entries

- Candidates include in their ***RequestVote*** RPCs information about the state of their log
- Before voting for a candidate, servers check that the log of the candidate is at least as up to date as their own log.
  - Majority rule does the rest
- Definition of Up-To-Date: next slide

Which log is more up to date?

- Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs.
  - If the logs have last entries with different terms, then the log with the later term is more up-to-date.
  - If the logs end with the same term, then whichever log is longer is more up-to-date.
- A log entry for a term means a leader was elected by a majority, and (inductively) earlier log records are up to date

New leader will not erase committed entries

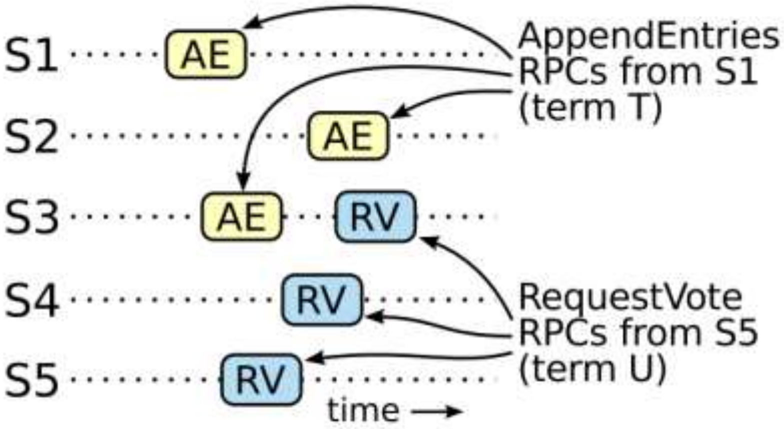
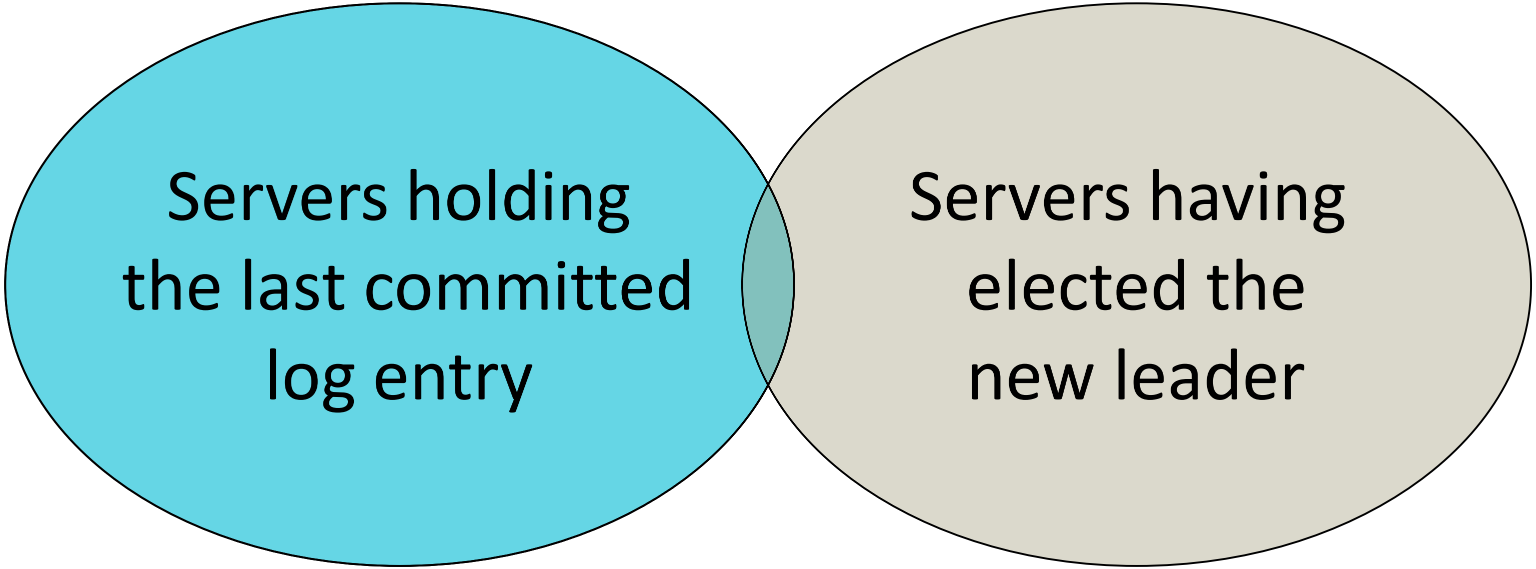


Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

Election restriction (II)



Two majorities of the same cluster ***must*** intersect

Committing entries from previous term

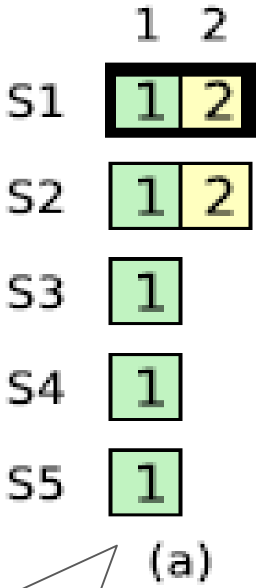
A leader cannot conclude that an entry from a previous term is committed even if stored on a majority of servers.

Leader should never commits log entries from previous terms by counting replicas

Should only do it for entries from the current term

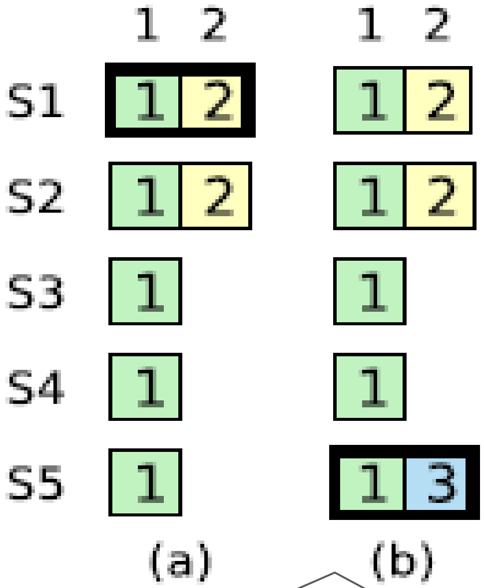
Once it has been able to do that for one entry, all prior entries are committed indirectly

Committing entries from previous term



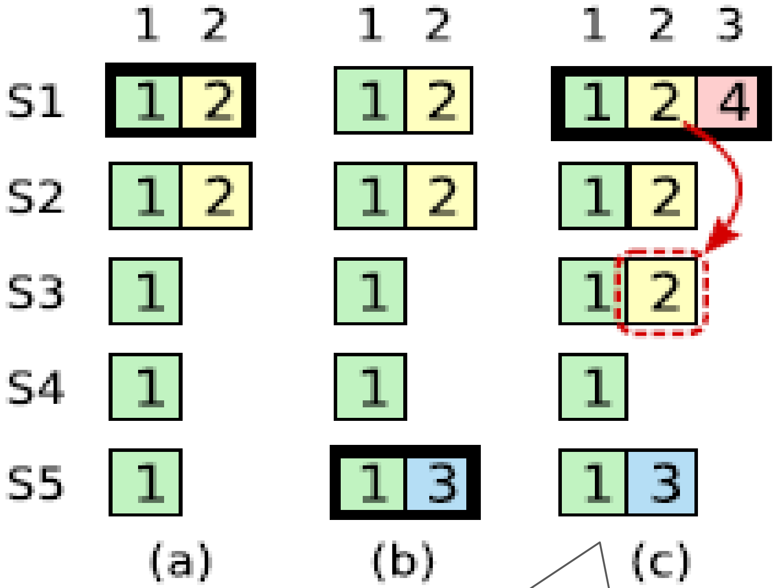
S1 is leader and partially replicates the log entry at index 2.

Committing entries from previous term



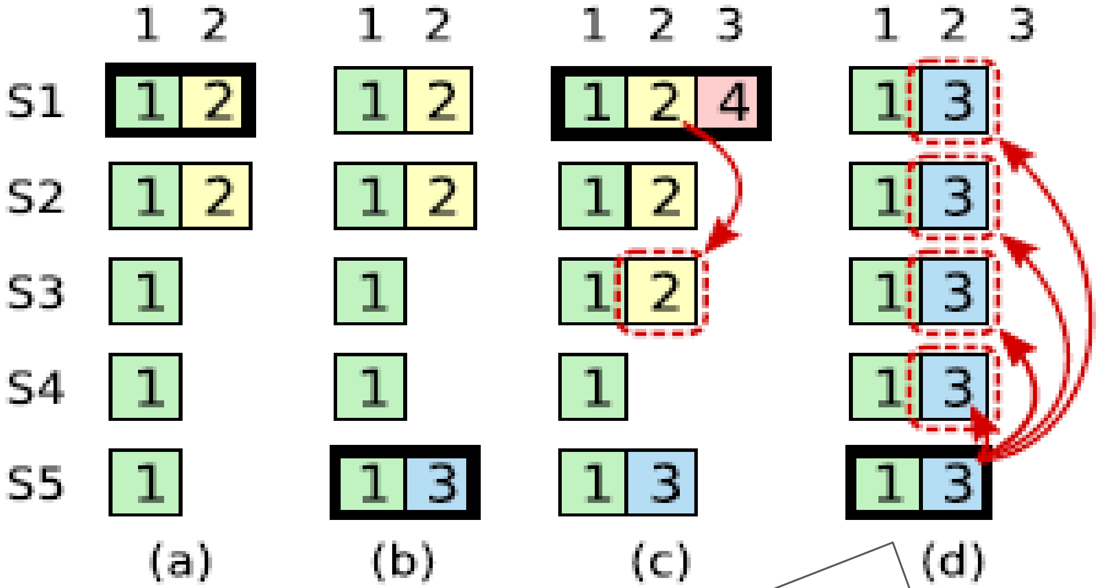
S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.

Committing entries from previous term



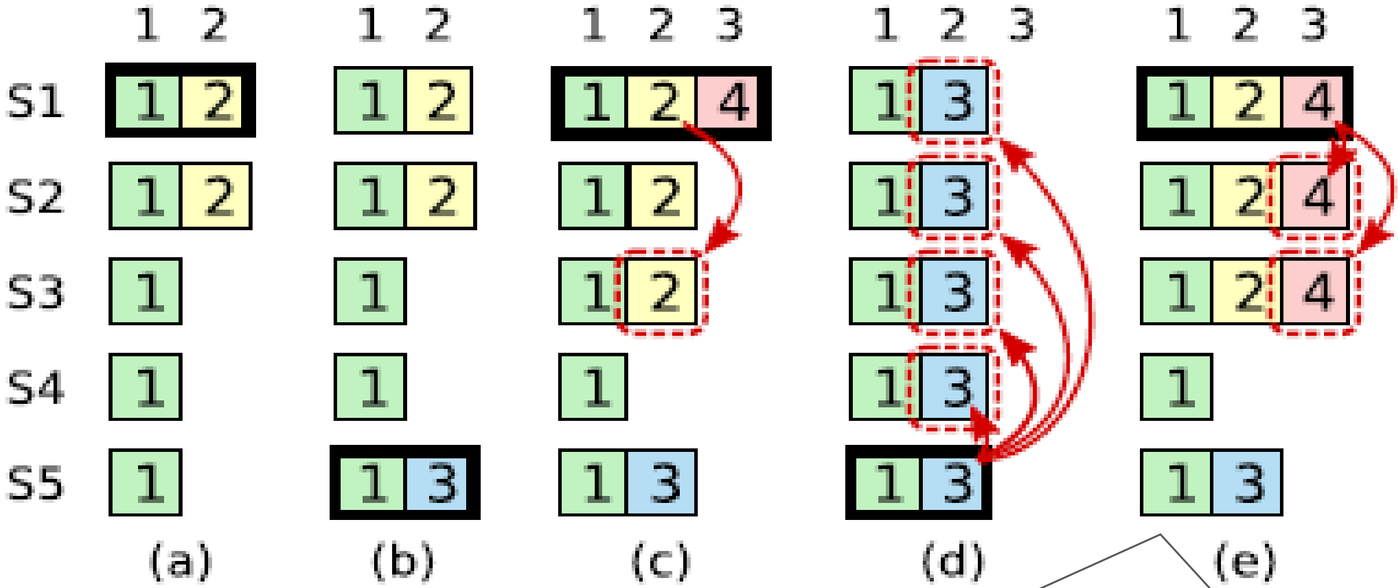
S5 crashes; S1 restarts, is elected leader, and continues replication

Committing entries from previous term



S1 crashes, S5 is elected leader (with votes from S2, S3, and S4) and overwrites the entry with its own entry from term 3.

Committing entries from previous term



However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as this entry is committed (S5 cannot win an election).

Explanations

In (a) S1 is leader and partially replicates the log entry at index 2.

In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.

In (c) S5 crashes; S1 restarts, is elected leader, and continues replication.

- Log entry from term 2 has been replicated on a majority of the servers, but it is not committed.

Explanations

If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3.

However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election).

At this point all preceding entries in the log are committed as well.

Cluster membership changes

Not possible to do an atomic switch

- Changing the membership of all servers at one

Will use a two-phase approach:

- Switch first to a transitional ***joint consensus*** configuration
- Once the joint consensus has been committed, transition to the new configuration

The joint consensus configuration

- Log entries are transmitted to all servers, old and new
- Any server can act as leader
- Agreements for entry commitment and elections requires majorities from both old and new configurations
- Cluster configurations are stored and replicated in special log entries

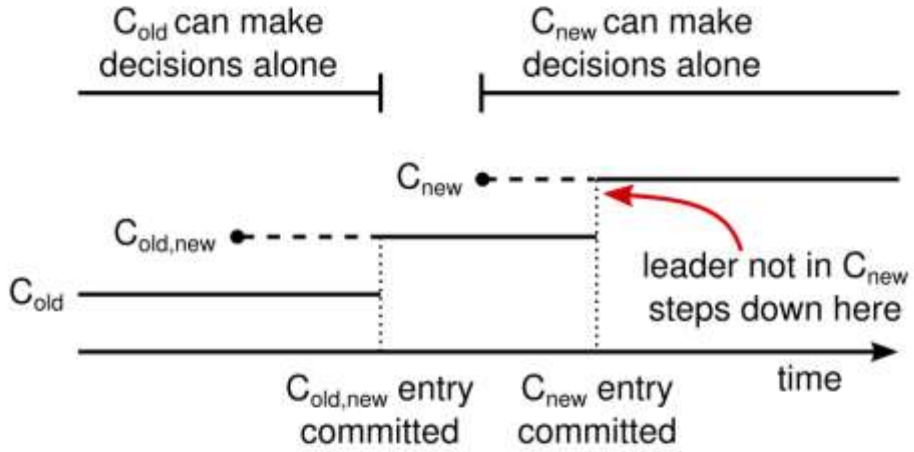
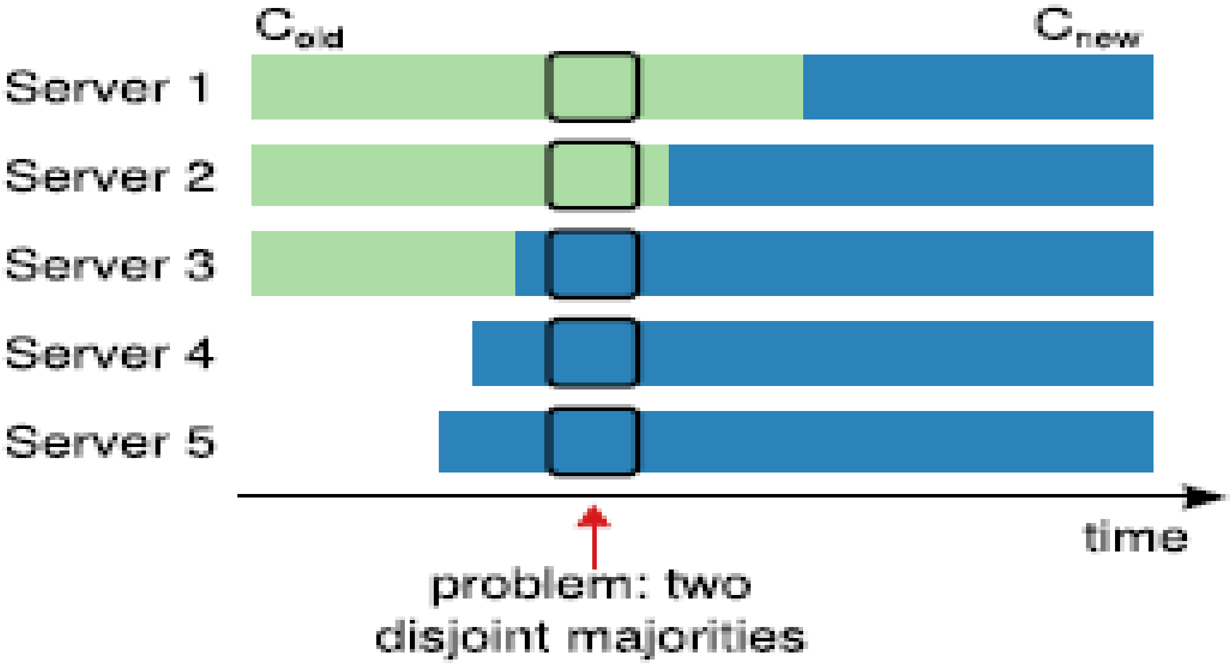


Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{old,new}$ configuration entry in its log and commits it to $C_{old,new}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

The joint consensus configuration



Implementations

- Two thousand lines of C++ code, not including tests, comments, or blank lines.
- About 25 independent third-party open source implementations in various stages of development
- Some commercial implementations
- **Automated formal proof of correctness**

Performance

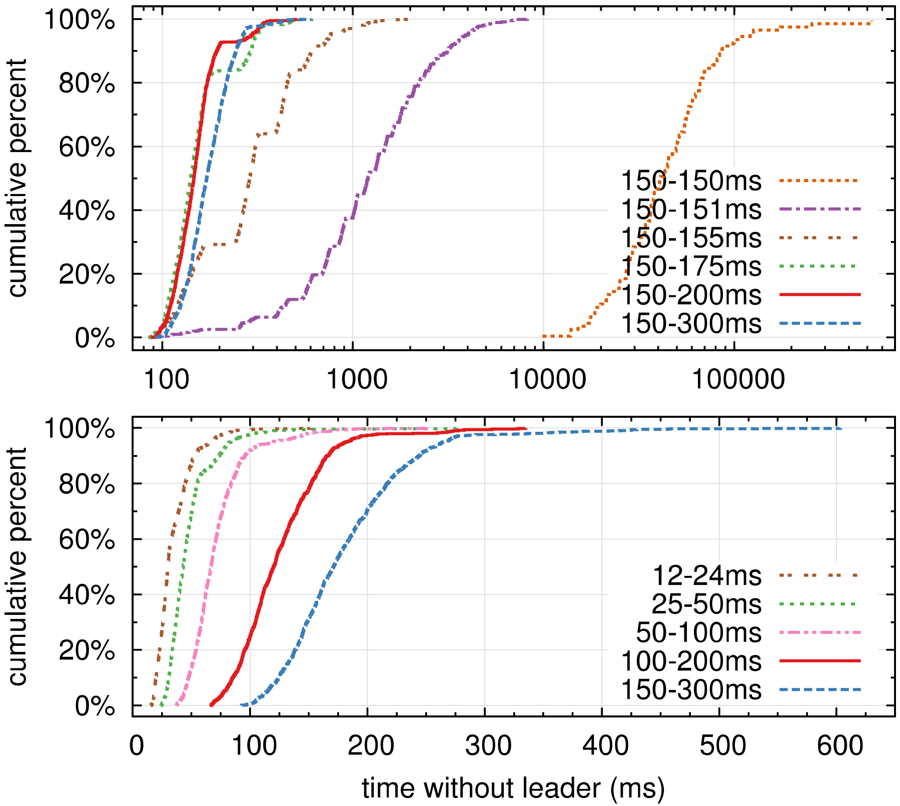


Figure 14: The time to detect and replace a crashed leader.

Related Work

- Paxos, multi Paxos and friends
    - And implementations such as Chubby, Spanner, ..
- ZooKeeper
- Viewstamped Replication
    - Raft has similarities with Zookeeper and VR in that they all elect a leader, and the leader ensures replication in order
    - But has simpler protocol for dealing with log conflicts
    - And simpler election protocol

User Studies

- Show that Raft is easier to understand than Paxos
  - Via quiz, etc

Summary

Consensus key building block in distributed systems

Raft similar to Paxos

Raft arguably easier to understand than Paxos

- It separates stages which reduces the algorithm state space
- Provides a more detailed implementation