In Search of an Understandable Consensus Algorithm

Diego Ongaro
John Ousterhout
Stanford University

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)

- ***Paxos***
  - Current standard for both teaching and implementing consensus algorithms
  - Very difficult to ***understand*** and very hard to ***implement***
- ***Raft***
  - New protocol (2014)
  - Much easier to ***understand***
  - Several ***open-source implementations***

Key features of Raft

- ***Strong leader:***
  - Leader does most of the work:
    - Issues ***all*** log updates
- ***Leader election:***
  - Uses ***randomized timers*** to elect leaders.
- ***Membership changes:***
  - New ***joint consensus*** approach where the majorities of two different configurations are required

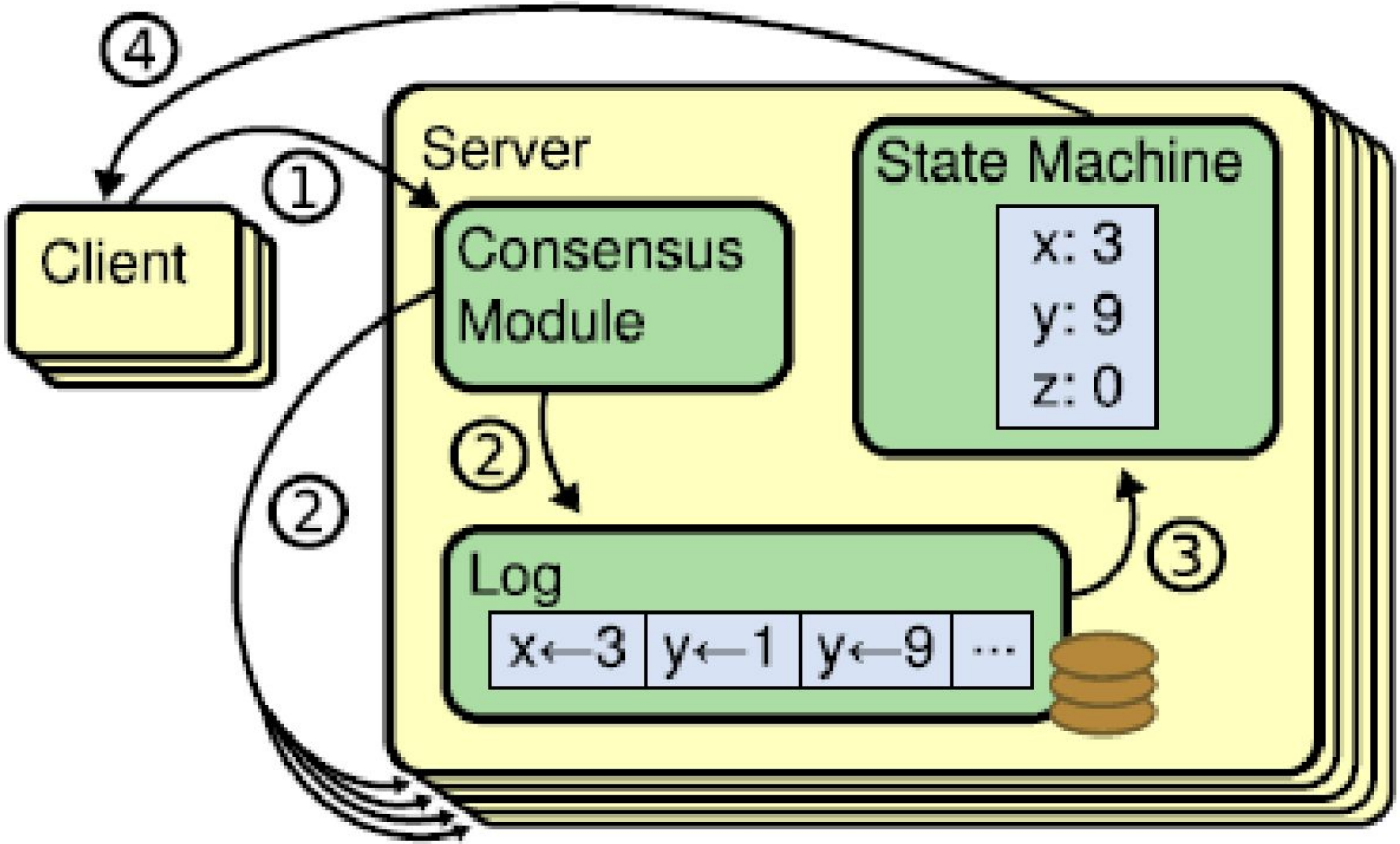
Replicated state machines

- 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

The distributed log (I)

- Each server stores a log containing commands
- Consensus algorithm ensures that all logs contain the ***same commands*** in the same order
- State machines always execute commands ***in the log order***
  - They will remain consistent as long as command executions have ***deterministic results***

The distributed log (II)



The distributed log (III)

- Client sends a command to one of the servers
- Server adds the command to its log
- Server forwards the new log entry to the other servers
- Once a consensus has been reached, each server state machine process the command and sends it reply to the client

Consensus algorithms (I)

- Typically satisfy the following properties
  - ***Safety:***
    - Never return an incorrect result under all kinds of non-Byzantine failures
  - ***Availability:***
    - Remain available as long as a majority of the servers remain operational and can communicate with each other and with clients.

Two types of failures

- ***<u>Non-Byzantine</u>***
  - Failed nodes stop communicating with other nodes
    - "Clean" failure
    - ***Fail-stop*** behavior
- ***<u>Byzantine</u>***
  - Failed nodes will keep sending messages
    - Incorrect and potentially misleading
    - Failed node becomes a ***traitor***

Consensus algorithms (II)

- ***Robustness:***
  - Do not depend on timing to ensure the consistency of the logs
- ***Responsiveness:***
  - Commands will typically complete as soon as a majority of the servers have responded to a ***single round*** of remote procedure calls
    - One or two slow servers will not impact overall system response times

Paxos limitations (I)

- Exceptionally difficult to understand

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

*The USENIX Symposium on Networked Systems Design and Implementation

Paxos limitations (II)

- Very difficult to implement

  "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

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
    - Could logs have holes in them? No

Problem decomposition

- Old technique
- René Descartes' third rule for avoiding fallacies:

    The third, to conduct my thoughts in such order that, by commencing with objects the simplest and easiest to know, I might ascend by little and little, and, as it were, step by step, to the knowledge of the more complex

Raft consensus algorithm (I)

- Servers start by electing a ***leader***
  - Sole server habilitated to accept commands from clients
  - Will enter them in its log and forward them to other servers
  - Will tell them when it is safe to apply these log entries to their state machines

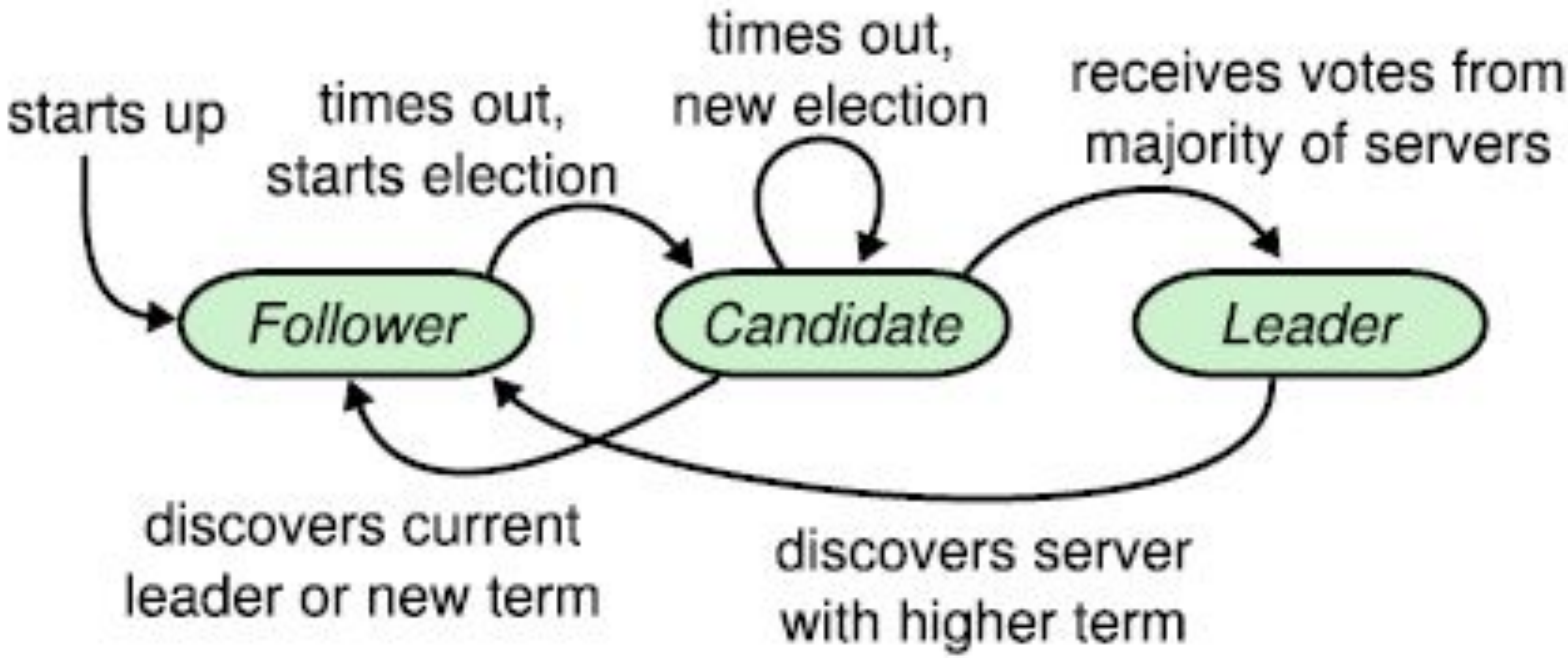
Raft consensus algorithm (II)

- Decomposes the problem into three fairly independent subproblems
  - ***Leader election:***
    How servers will pick a—***single***—leader
  - ***Log replication:***
    How the leader will accept log entries from clients, propagate them to the other servers and ensure their logs remain in a consistent state
  - ***Safety***

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

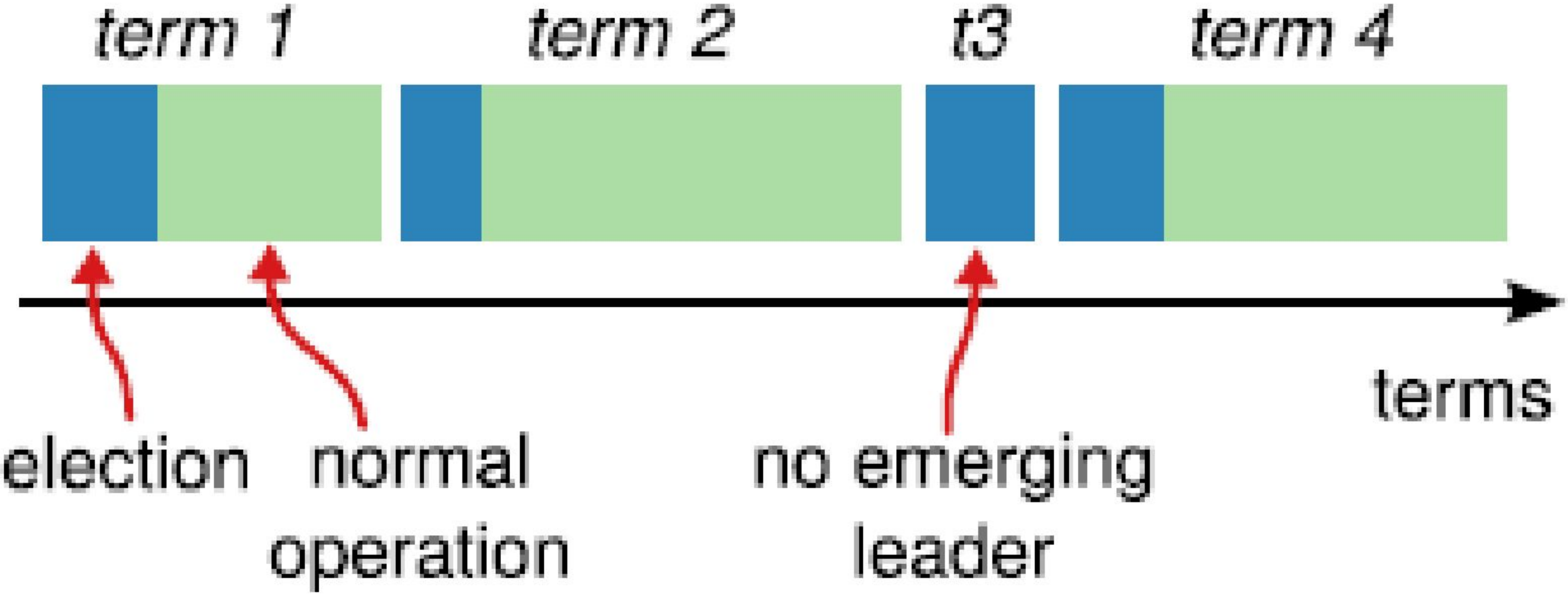
Server states



Raft basics: terms (I)

- Epochs of arbitrary length
  - Start with the election of a leader
  - End when
    - No leader can be selected (split vote)
    - Leader becomes unavailable
- Different servers may observe transitions between terms at different times or even miss them

Raft basics: terms (II)



Raft basics: terms (III)

- Terms act as logical clocks
  - Allow servers to detect and discard obsolete information (messages from stale leaders, …)
- Each server maintains a current term number
  - Includes it in all its communications
- A server receiving a message with a high number updates its own number
- A leader or a candidate receiving a message with a high number becomes a follower

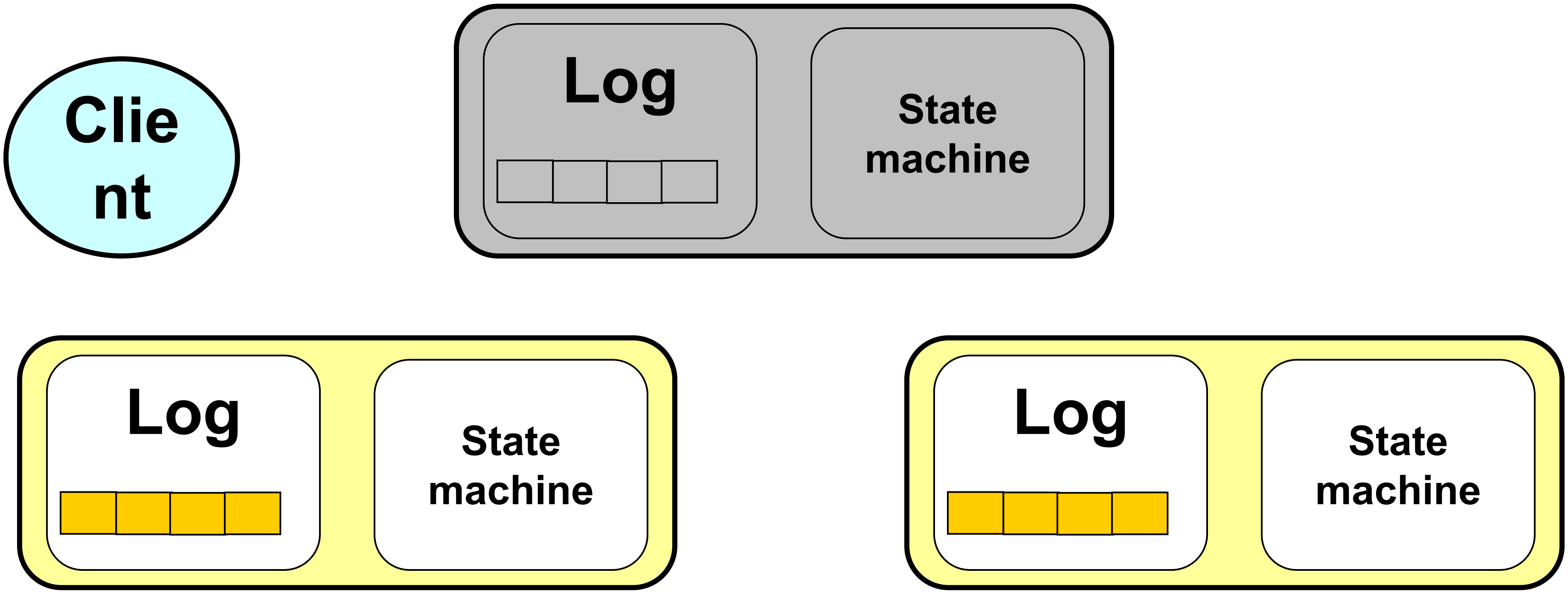
Raft basics: RPC

- 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

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

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

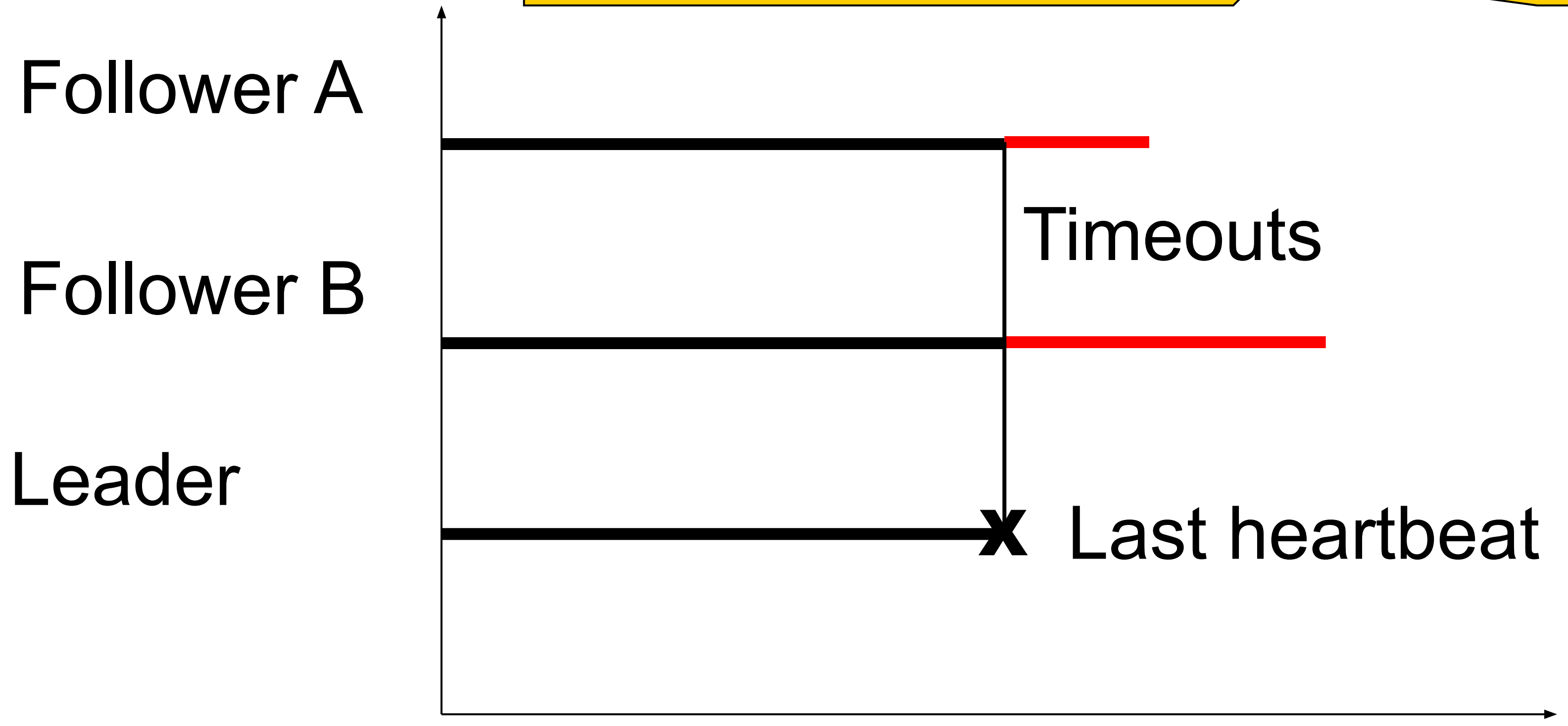
- No candidate obtains a majority of the votes in the servers in the cluster
- Each candidate will time out and start a new election
  - After incrementing its term number

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

Example

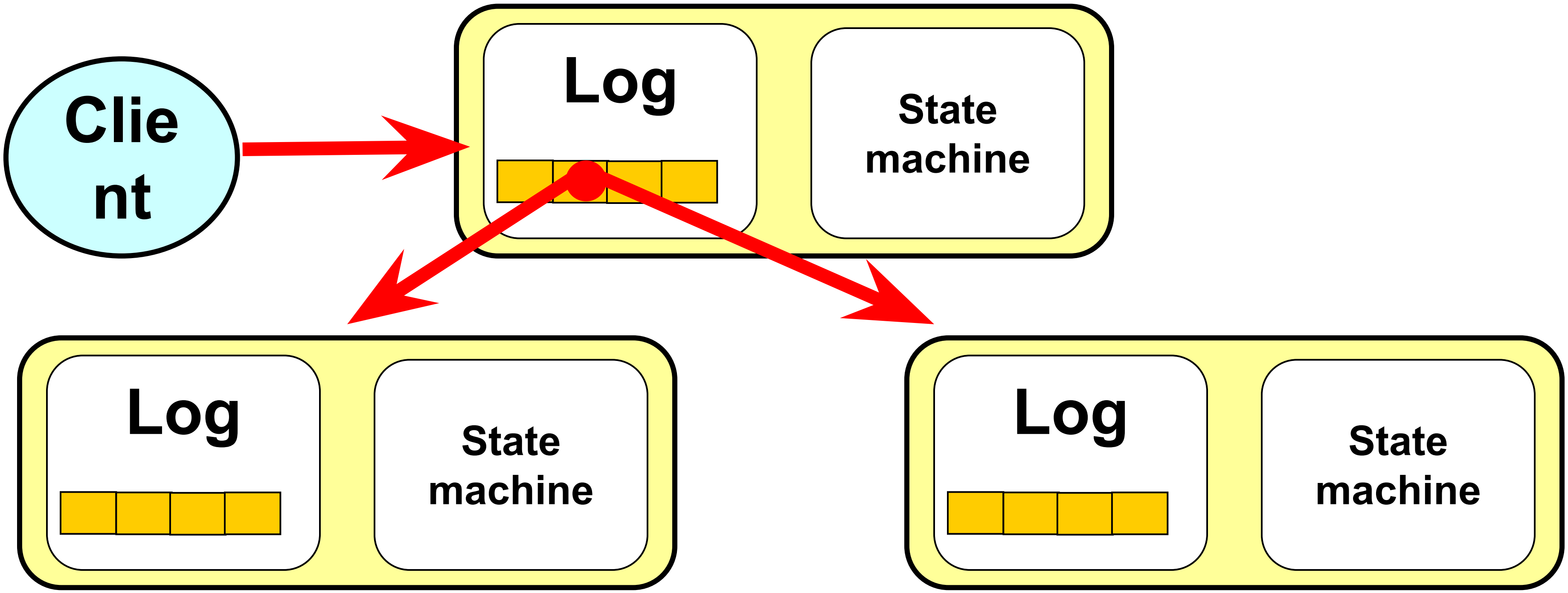
Follower with the ***shortest timeout*** becomes the ***new leader***



Log replication

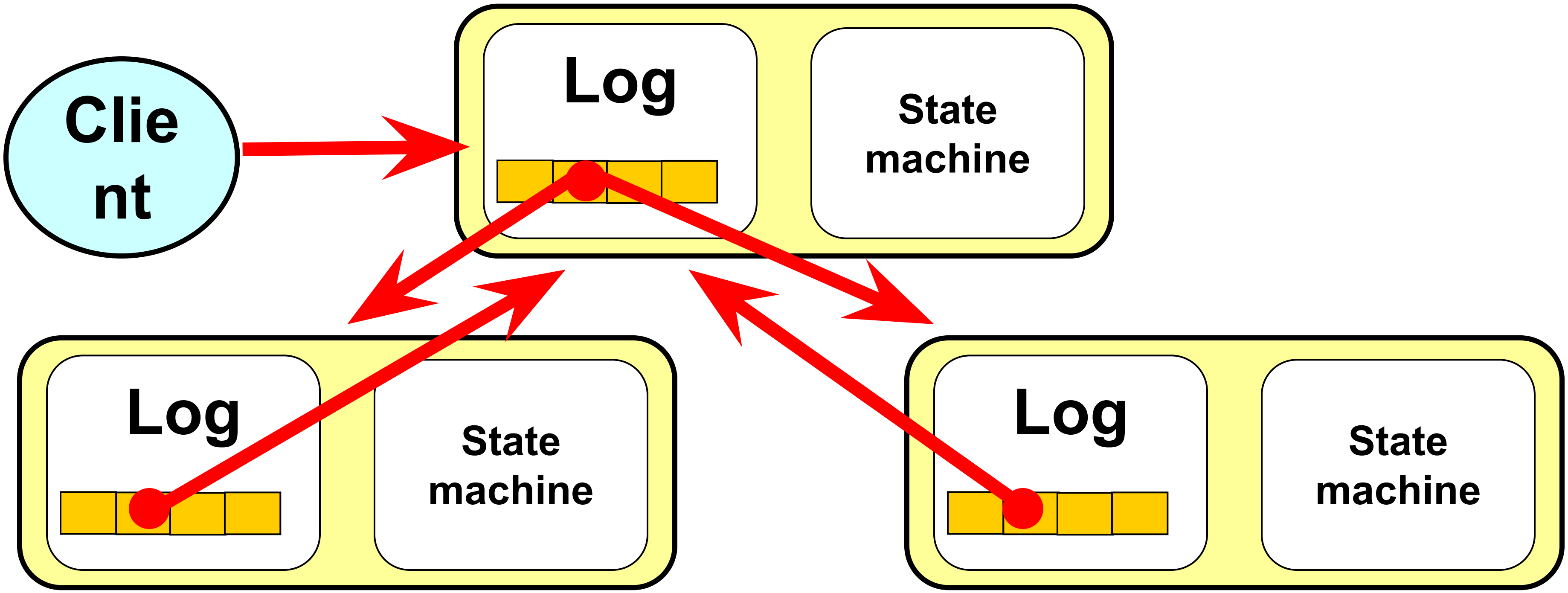
- Leaders
  - Accept client commands
  - Append them to their log (new entry)
  - Issue **AppendEntry** RPCs in parallel to all followers
  - Apply the entry to their state machine once it has been safely replicated
    - Entry is then ***committed***

A client sends a request



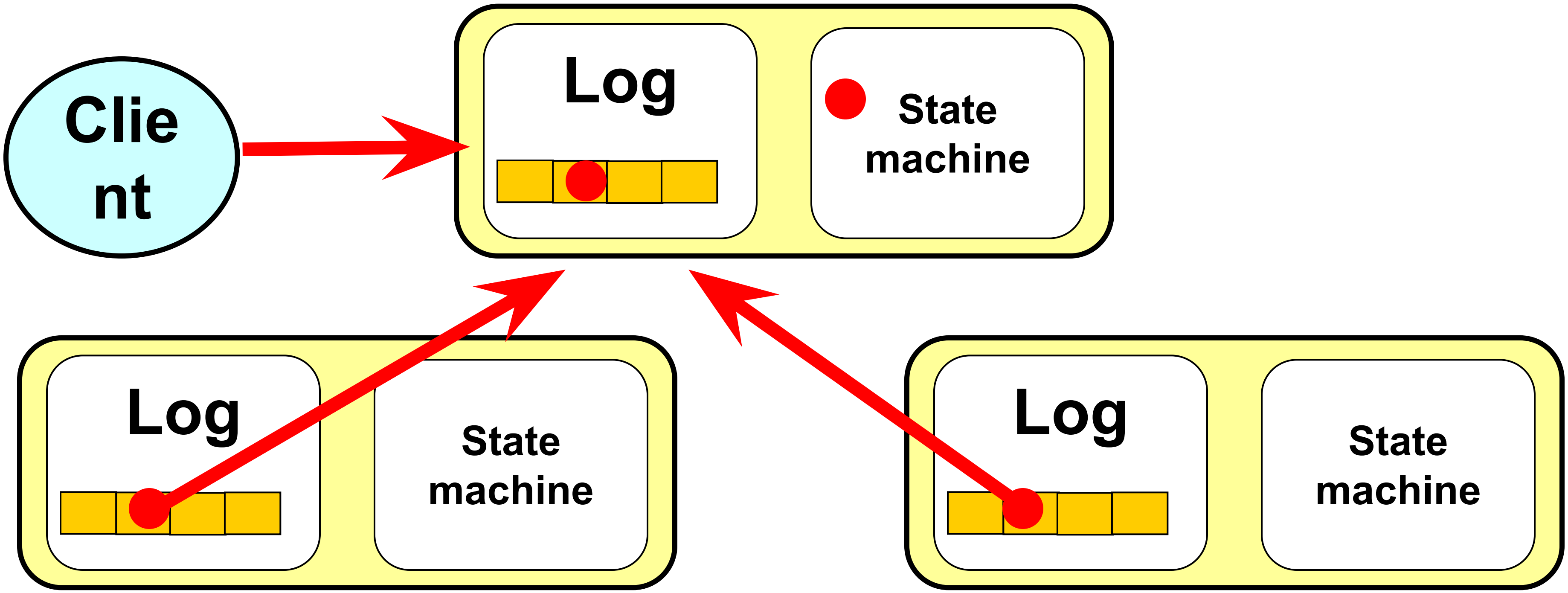
- Leader stores request on its log and forwards it to its followers

The followers receive the request



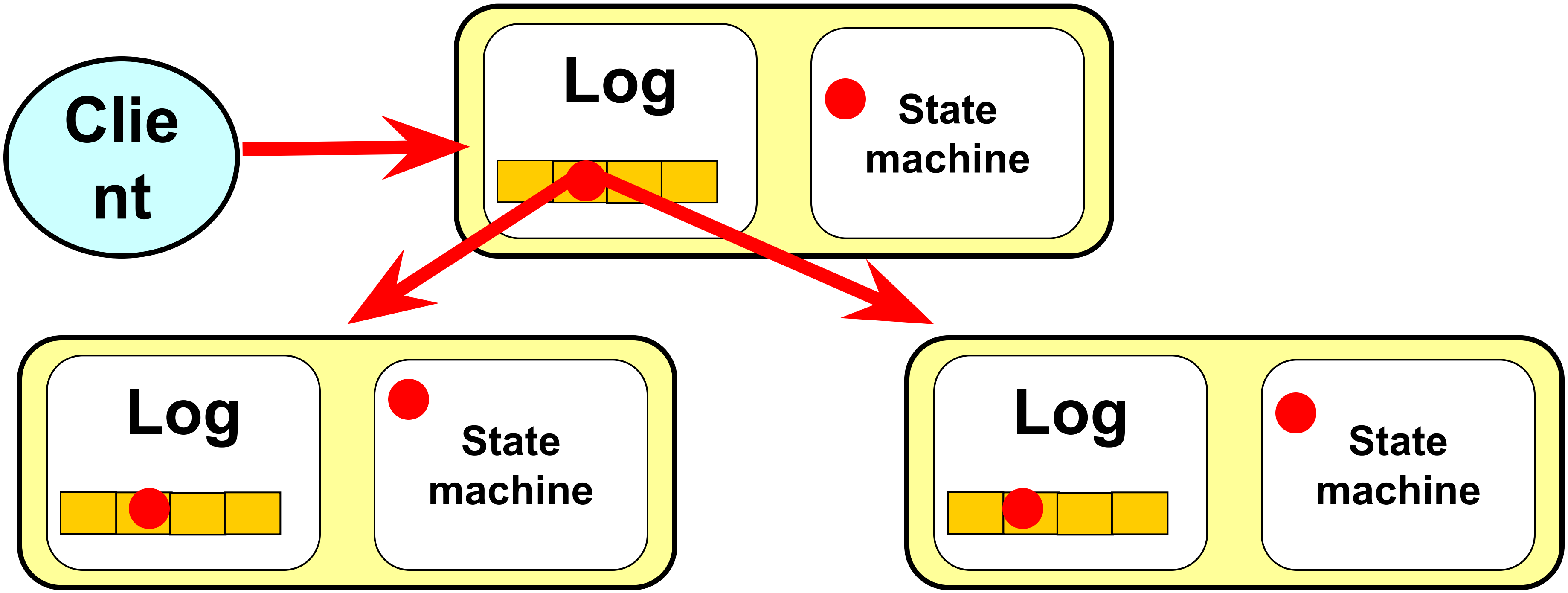
- Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



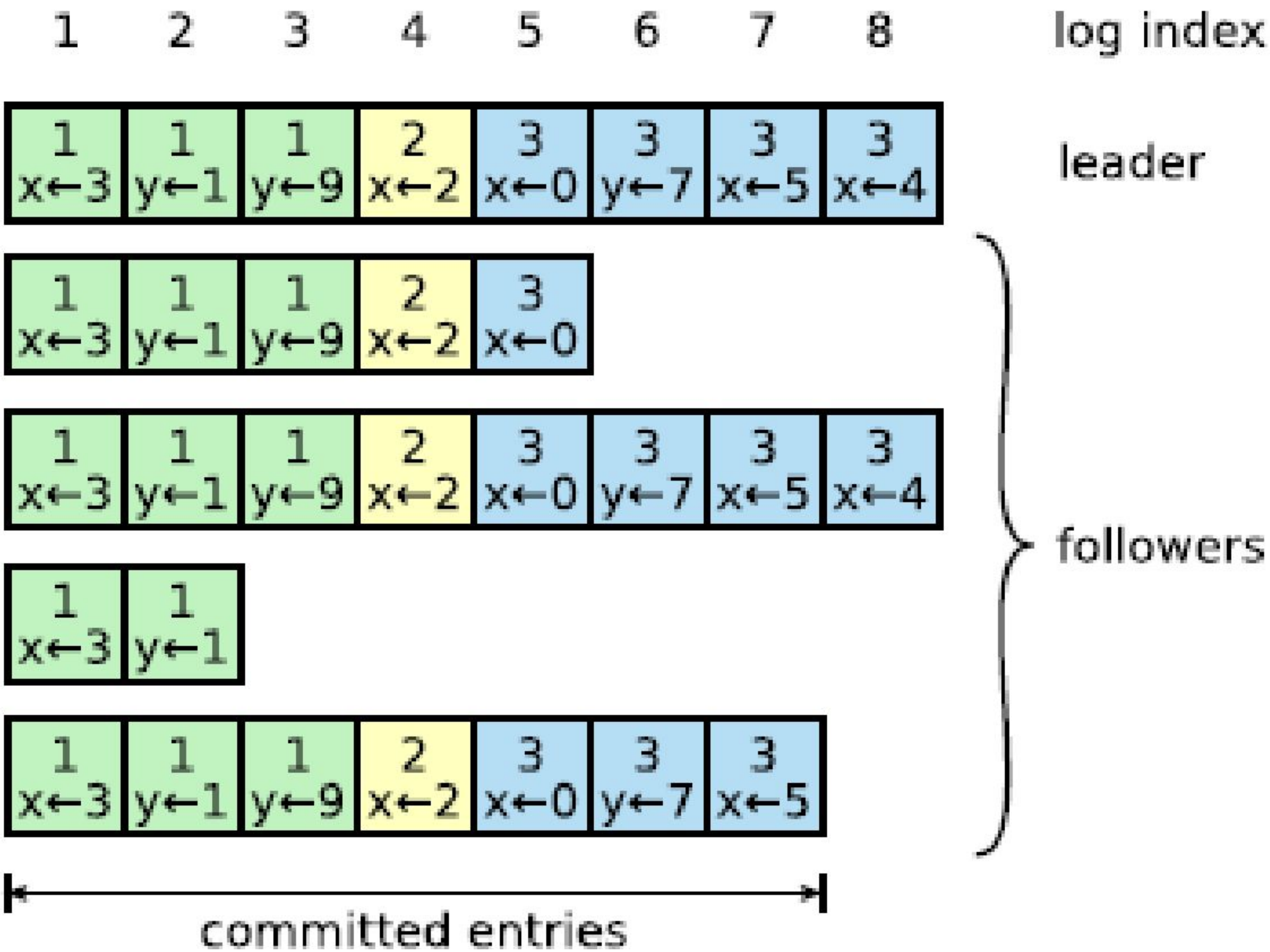
- Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



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

Log organization



Colors identify terms

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

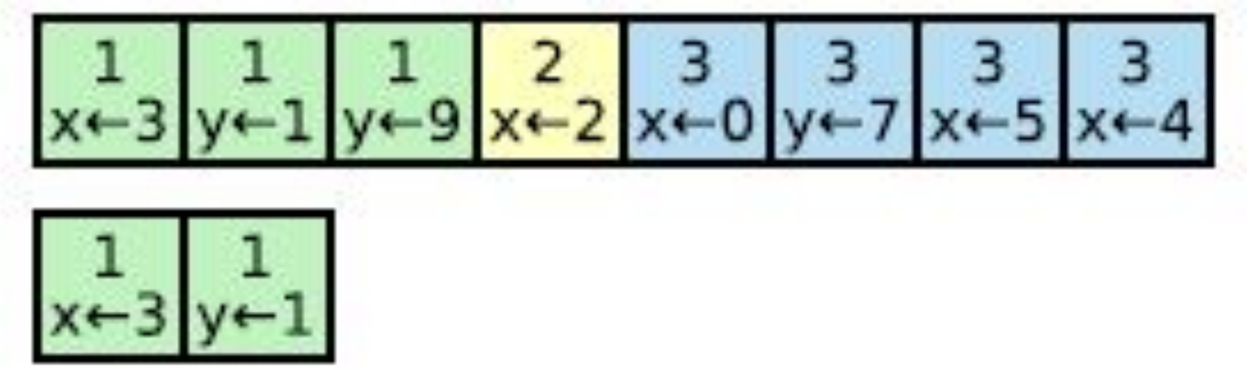
Why?

- 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

Raft log matching property

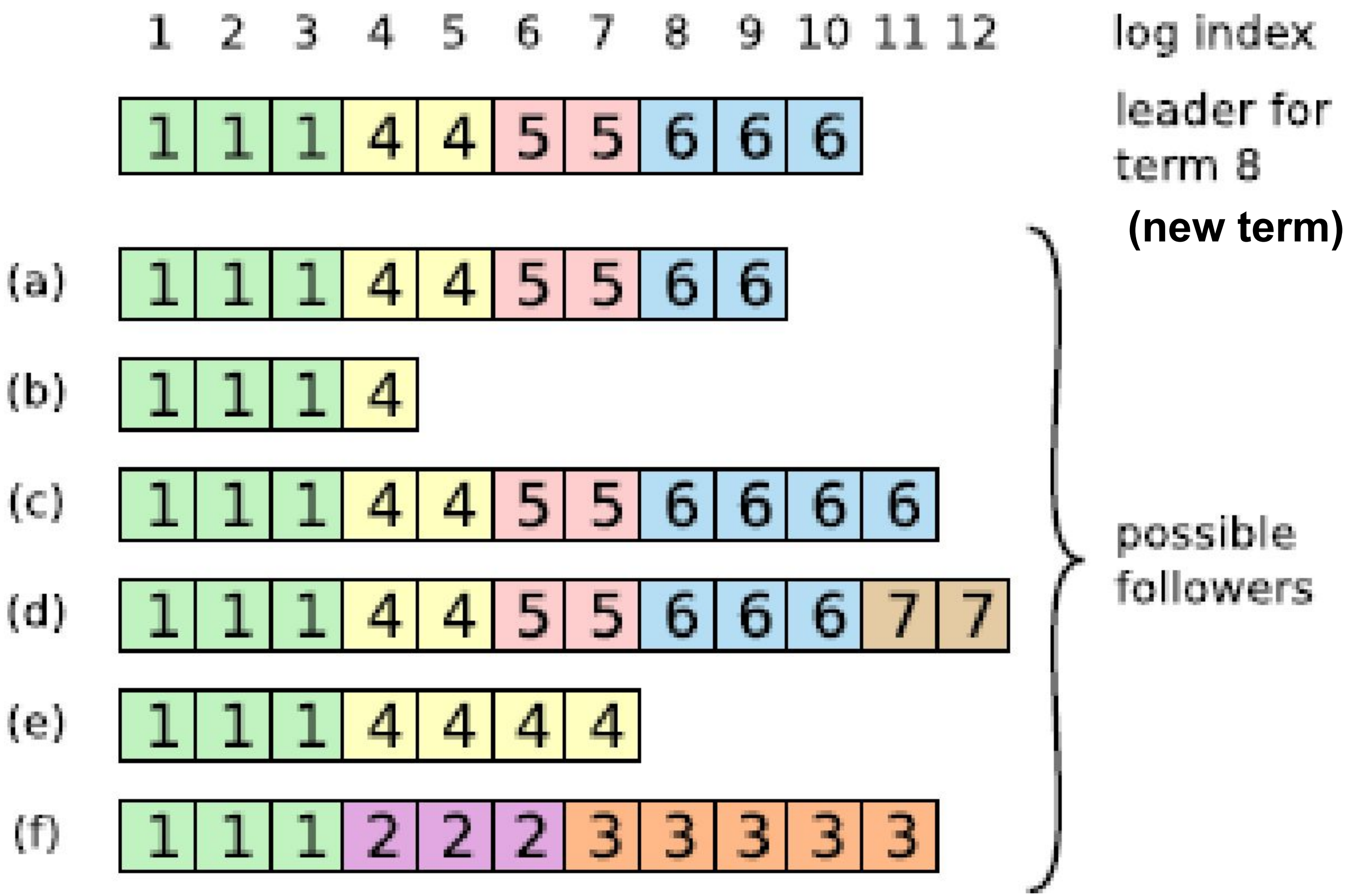
- If two entries in different logs have the same index and term
  - These entries store the same command
  - ***All previous entries*** in the two logs are ***identical***



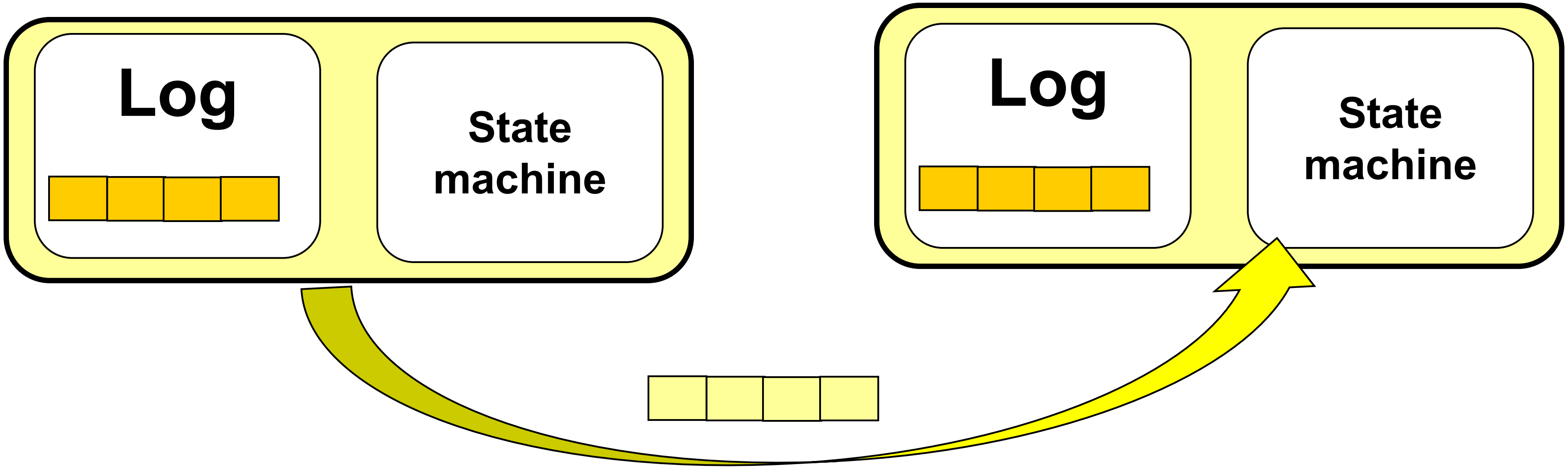
Handling leader crashes (I)

- 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)

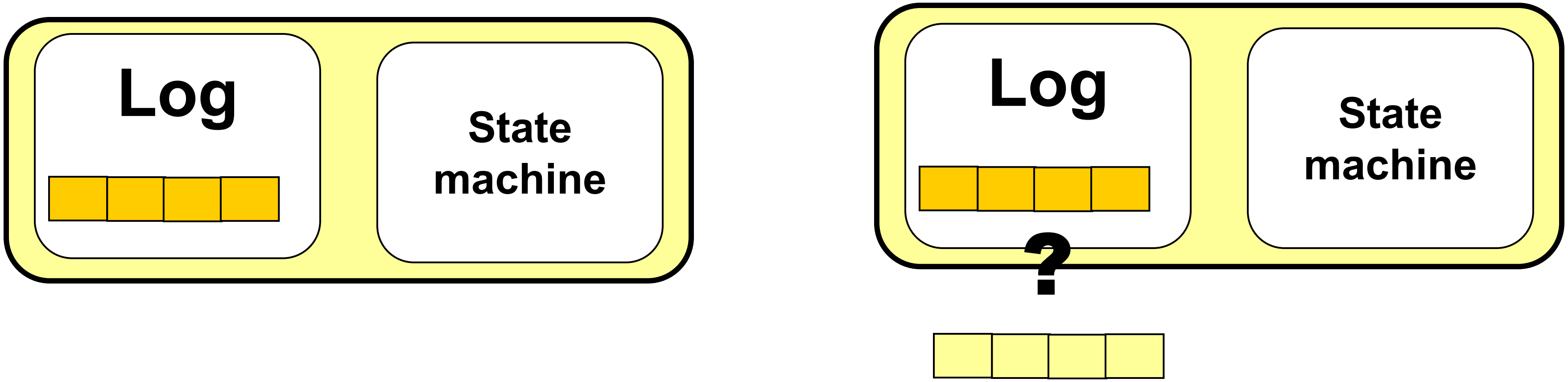


An election starts



- Candidate for leader position requests votes of other former followers
  - Includes a summary of the state of its log

Former followers reply

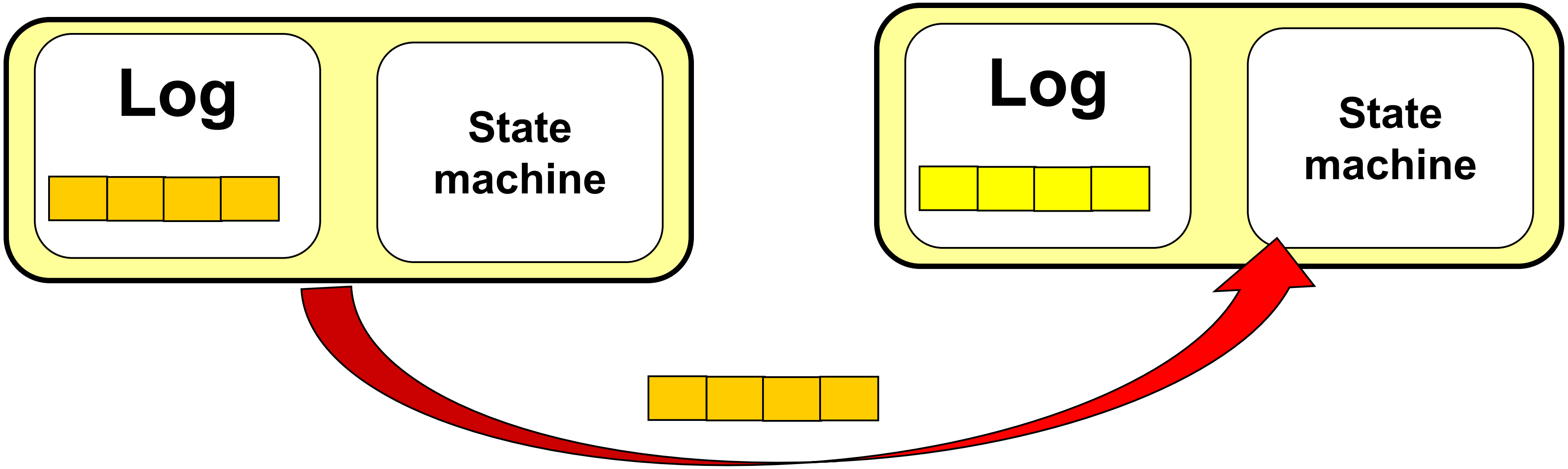


- Former followers compare the state of their logs with credentials of candidate
- Vote for candidate unless
  - Their own log is more "up to date"
  - They have already voted for another server

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

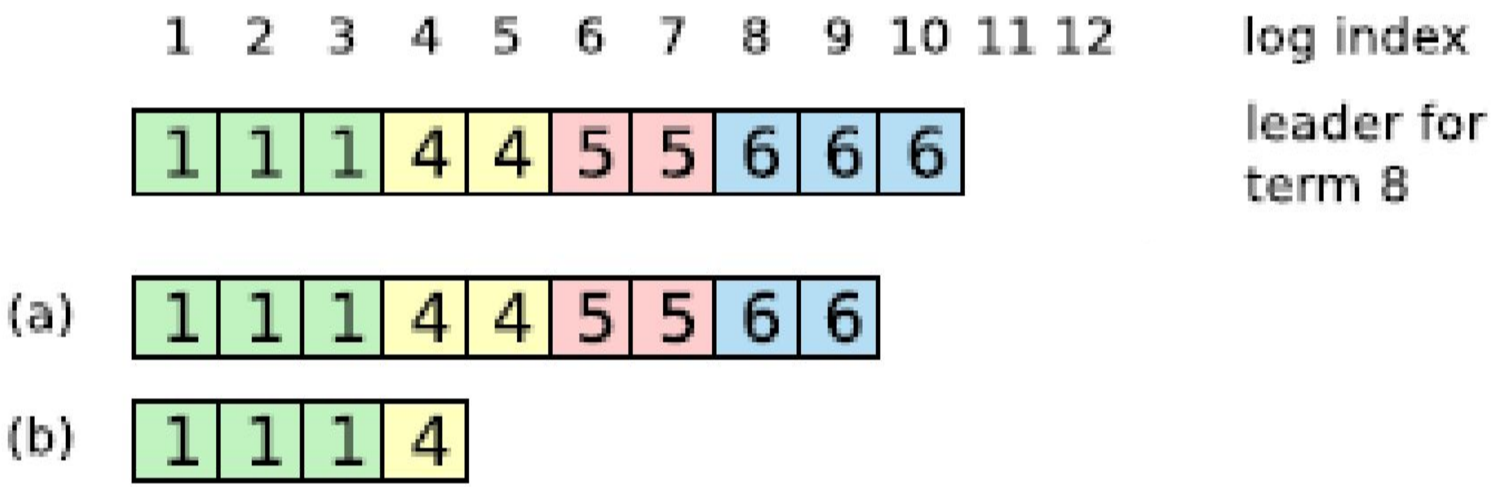
How? (I)

- 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

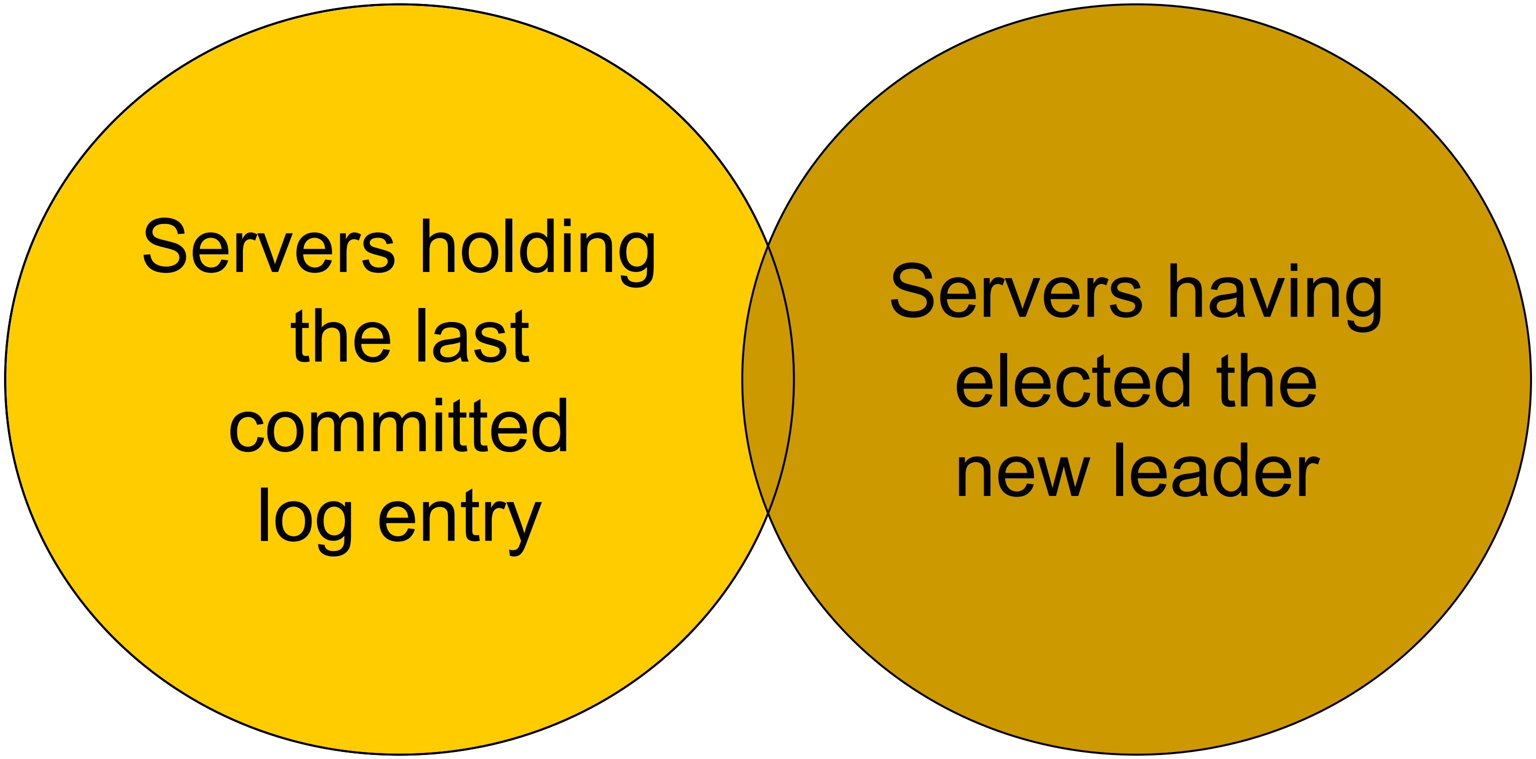
Safety

- Two main issues
  - What if the log of a new leader did not contain all previously committed entries?
    - Must impose conditions on new leaders
  - 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
    - *Details in the paper*
  - 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

Election restriction (II)

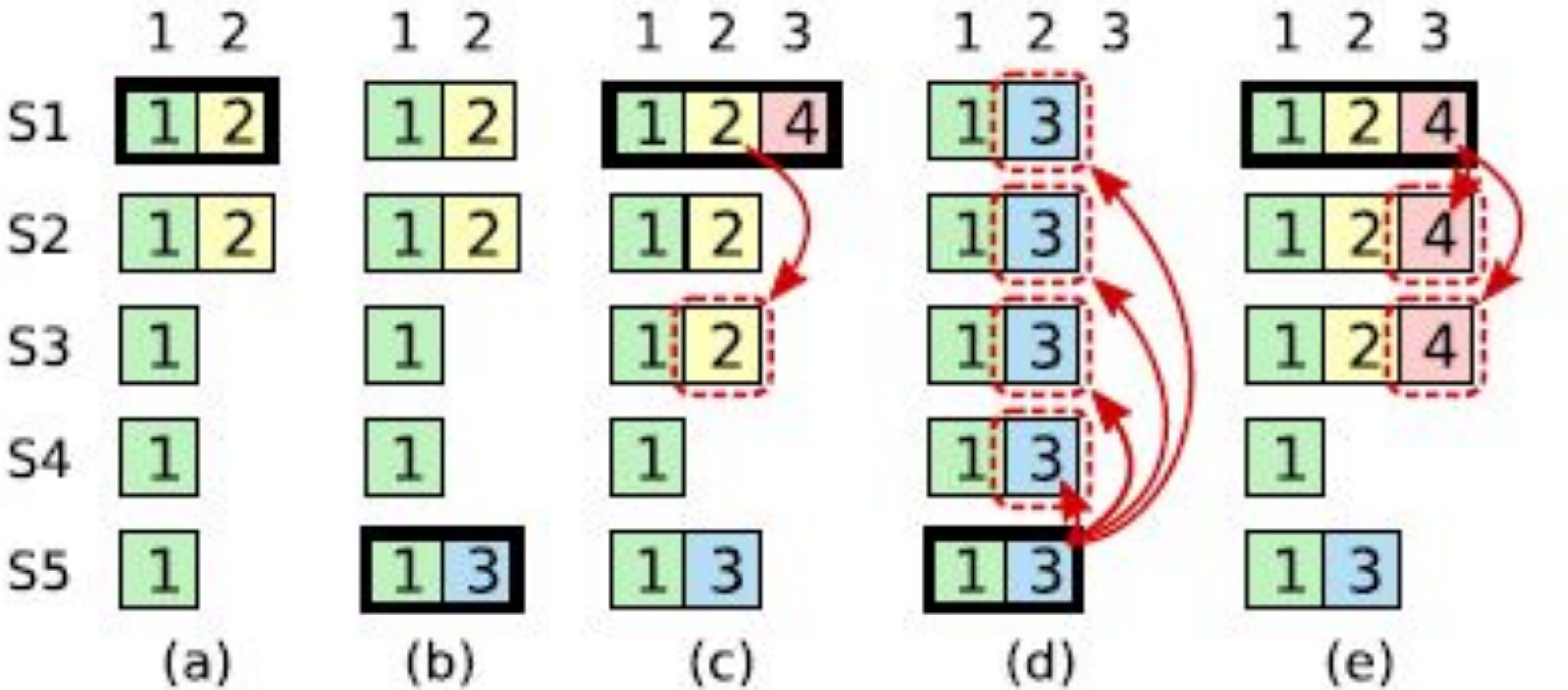


Two majorities of the same cluster ***must*** intersect

Committing entries from a previous term

- A leader cannot immediately conclude that an entry from a previous term even is committed even if it is stored on a majority of servers.
  - *See next figure*
- 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 a previous term



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.

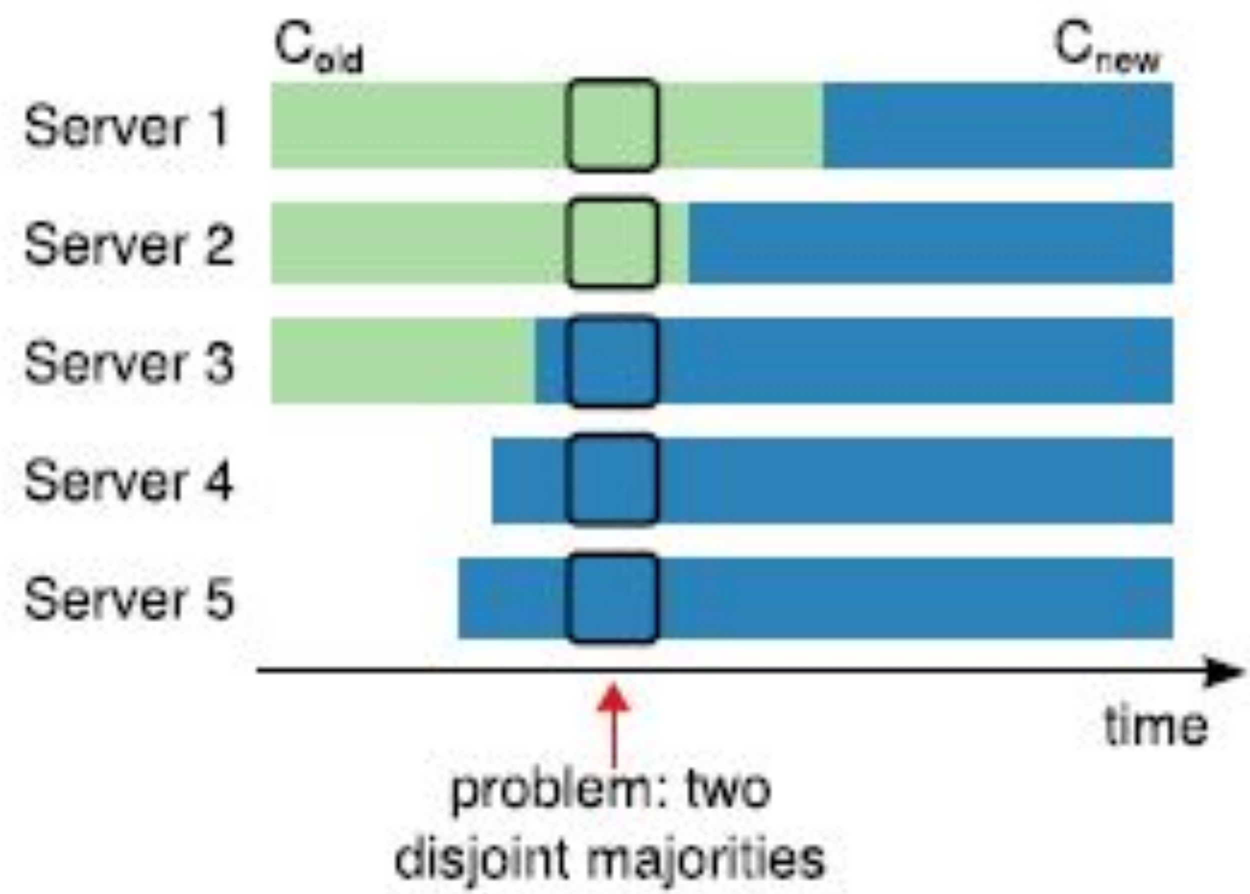
Cluster membership changes

- 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

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

Understandability

- See paper

Correctness

- A proof of safety exists

Performance

- See paper

Conclusion

- Raft is much easier to understand and implement than Paxos and has no performance penalty