In Search of an Understandable Consensus Algorithm

Ongaro, D. and Ousterhout, J. (2014)
Proceedings of the USENIX Annual Technical Conference, pp. 305–319.

https://www.youtube.com/watch?v=no5Im1daS-o

Motivation

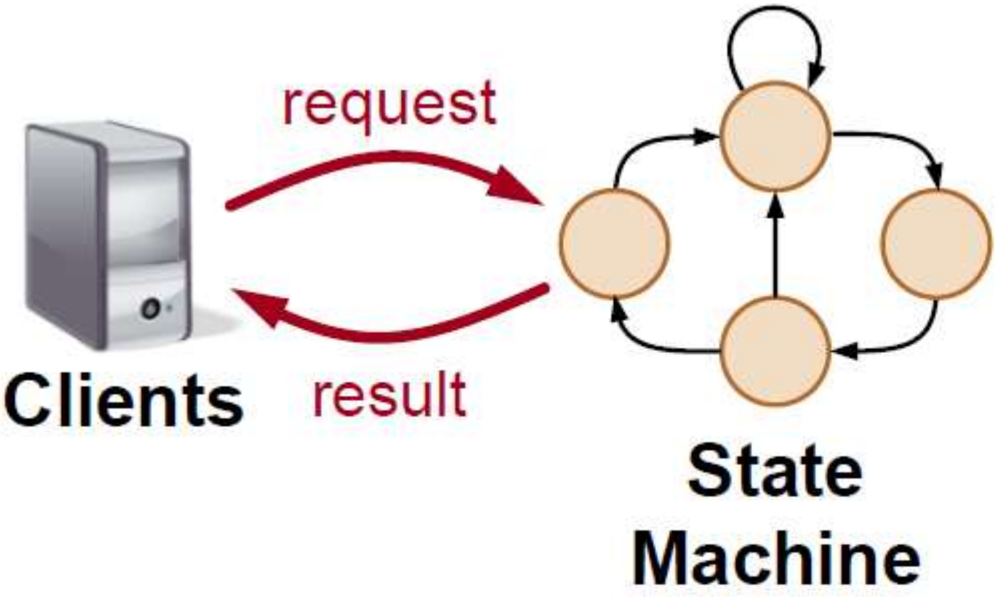
- Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members
  - Very important role in building fault-tolerant distributed systems
  - $n > 2f$

Motivation

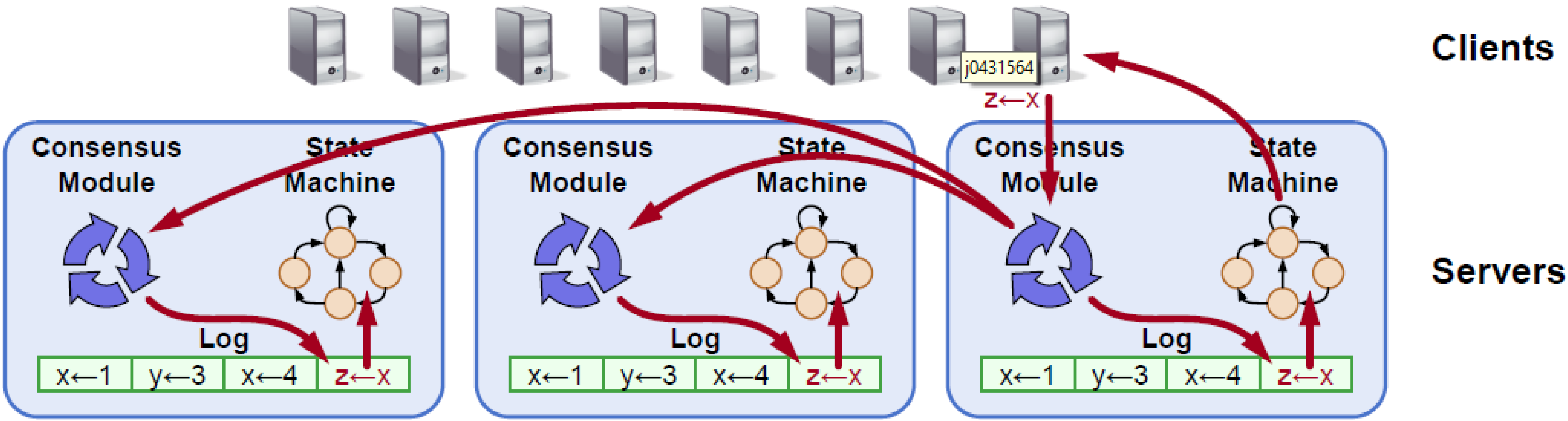
- Paxos
  - Current standard for both teaching and implementing consensus algorithms
  - Very difficult to understand and very hard to implement
- Raft
  - Much easier to understand
  - Several open-source implementations

State machine

- Responds to external stimuli
- Manages internal state



Replicated state machine



- Consensus module ensures proper log replication
- Replicated log ensures state machines execute same commands in same order

Paxos widely regarded as difficult

- Hard 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

Paxos widely regarded as difficult

- Not complete enough for real implementations

"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

Raft: Designed for understandability

- Main objective
  - Whenever possible, select the alternative that is the easiest to understand
- Techniques that were used include
  - Problem decomposition
  - Minimize state space
    - Handle multiple problems with a single mechanism
    - Eliminate special cases
      - Could logs have holes in them? No

Raft decomposition

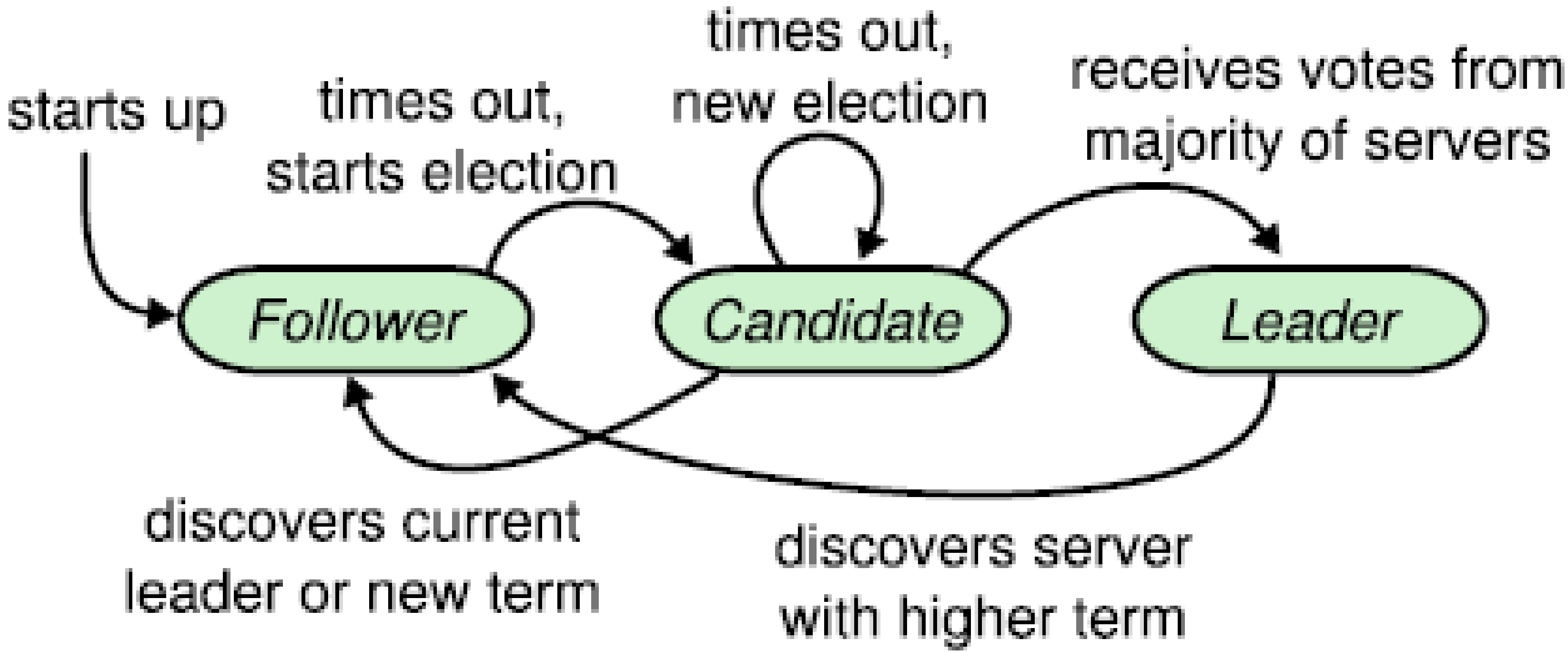
1. **Leader election**
   - Select one server to act as leader
   - Detect crashes, choose new leader

2. **Log replication (normal operation)**
   - Leader takes commands from clients, appends to its log
   - Leader replicates its log to other servers (overwriting inconsistencies)
   - Leader tells other servers when it is safe to apply log entries to their state machines

Raft decomposition

3. **Safety (when leader crashes)**
   - Keeps logs consistent
   - Only a server with an up-to-date log can become leader

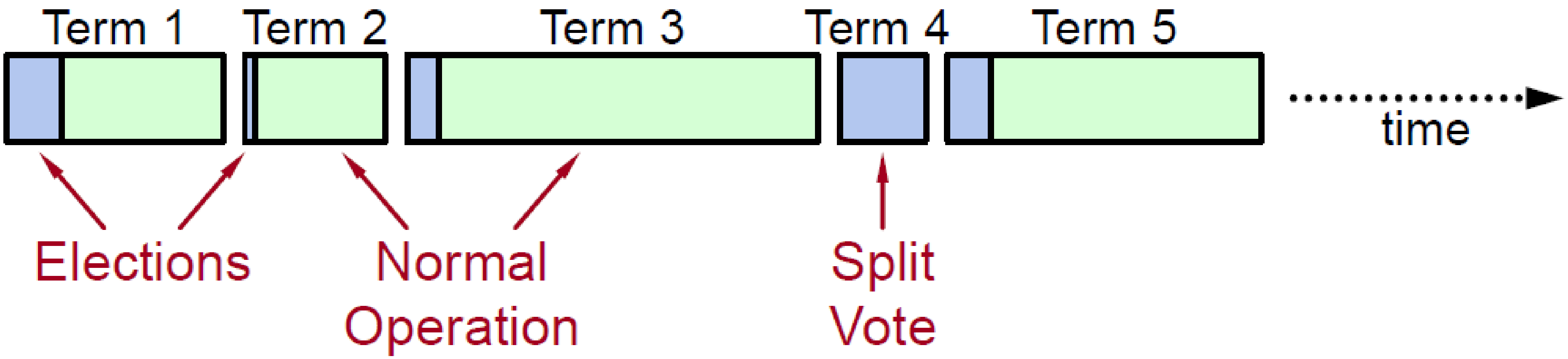
Raft basics: server states



Raft basics: RPCs

- Servers communicate though idempotent RPCs
  - **RequestVote**
    - Issued by a candidate to get elected as leader
  - **AppendEntries**
    - Issued by leader to
      - Replicate its log
      - Provide a form of heartbeat to maintain leadership

Raft basics: terms



- At most 1 leader per term
- Some terms have no leader (failed election)

Raft basics: terms

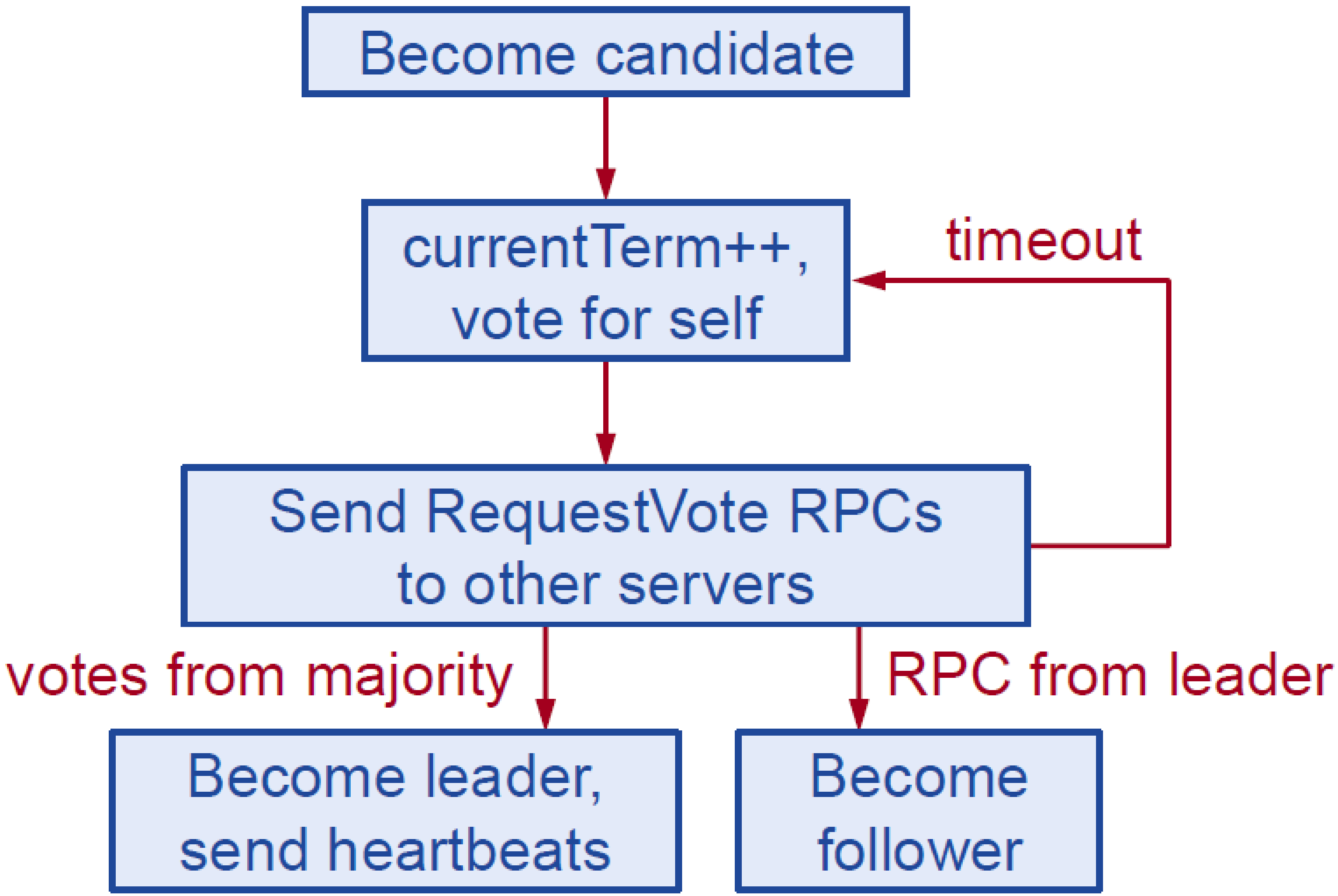
- Each server maintains current term value
  - Exchanged in every RPC
  - Peer has later term? Update term, revert to follower
  - Incoming RPC has obsolete term? Reply with error

- Terms identify obsolete information
  - Messages from stale leaders, …

Leader election

- 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 becomes candidate and starts an election to pick a new leader


Become candidate
currentTerm++,
vote for self
timeout
Send RequestVote RPCs
to other servers
votes from majority
RPC from leader
Become leader,
send heartbeats
Become
follower

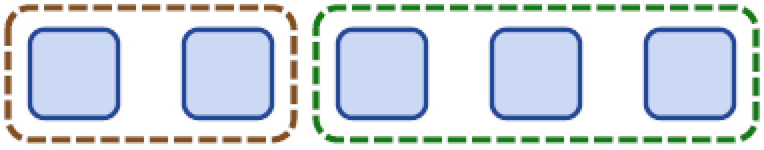
Voting

- Each server will vote for at most one candidate in a given term

- The first one that contacted it

Winning an election

- Must receive votes from a majority of the servers for the same term
- Majority rule ensures that at most one candidate can win an election per term



- Winner becomes leader and sends heartbeat messages to all other servers to assert its new role

Split elections

- No candidate obtains a majority of the votes
- Each candidate will time out and start a new election, after incrementing its term number

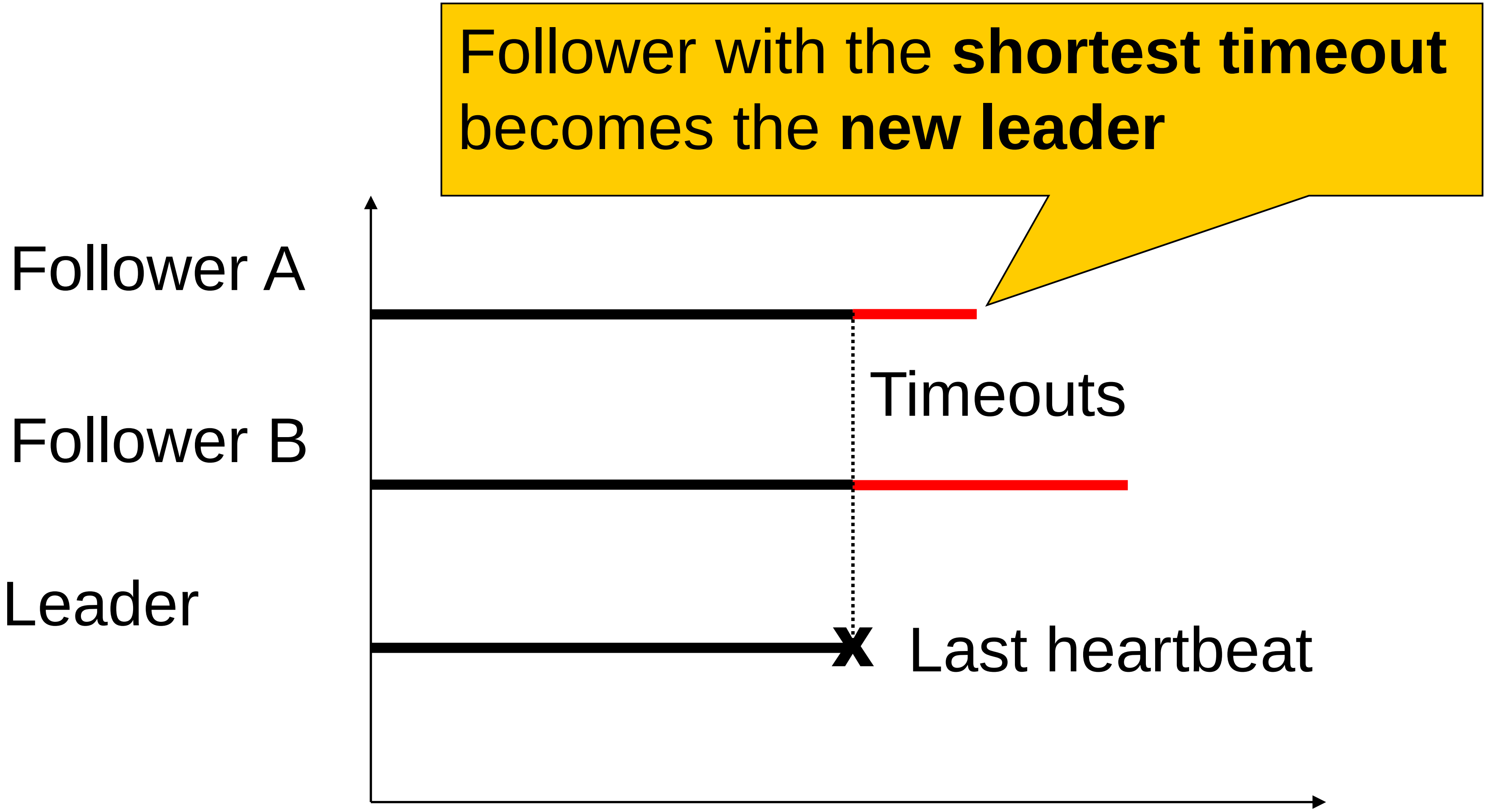
Avoiding split elections

- Raft uses **randomized** election timeouts
  - Chosen from a fixed interval

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

- Randomized approach simpler than ranking

Liveness


Follower with the **shortest timeout** becomes the **new leader**
Follower A
Timeouts
Follower B
Leader
x
Last heartbeat

Log replication

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to all followers

Commit

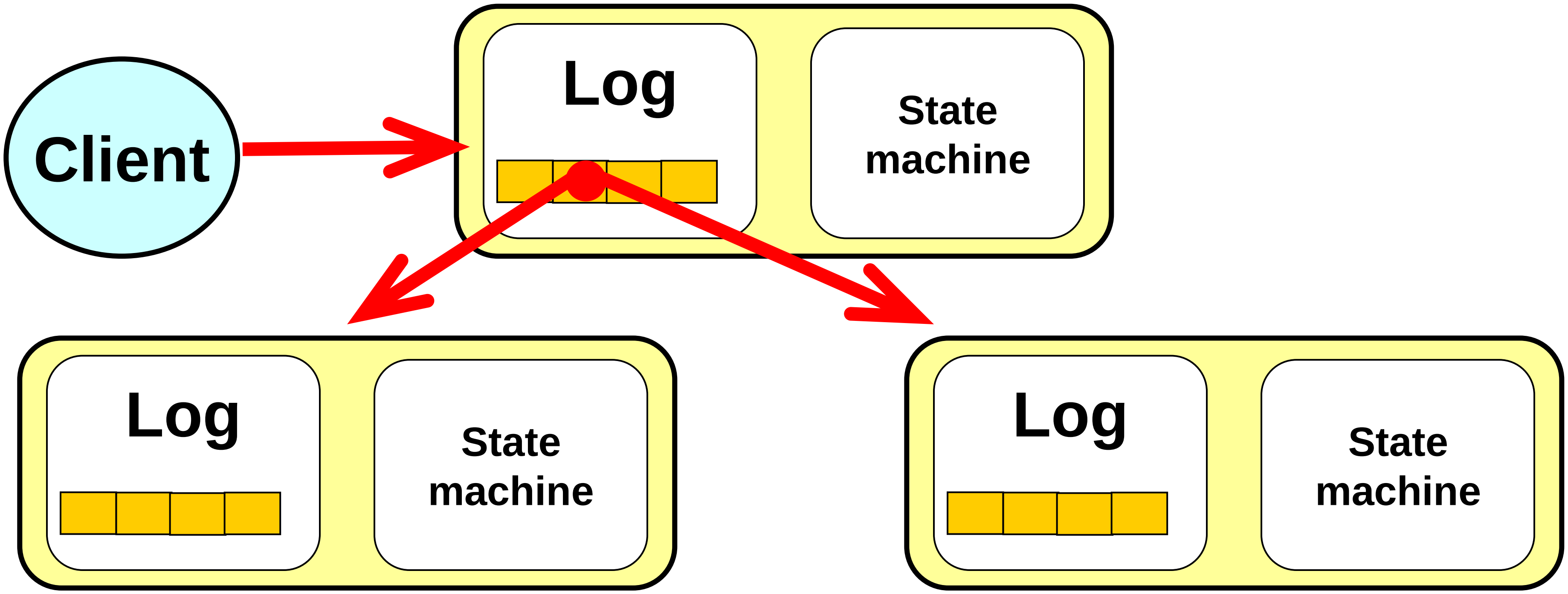
Optimal performance in common case:

- One successful RPC to any majority of servers

Normal operation

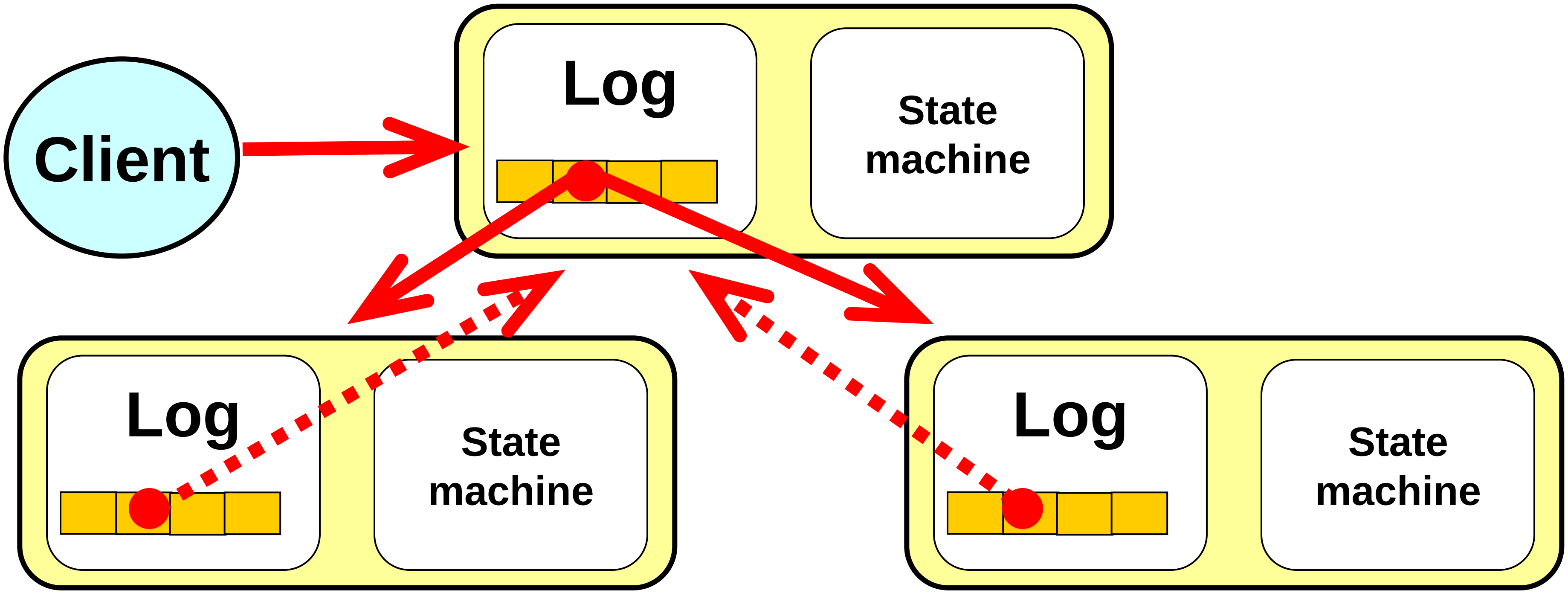
- Once new entry is commited:
  - Leader executes command in its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers execute committed commands in their state machines

A client sends a request



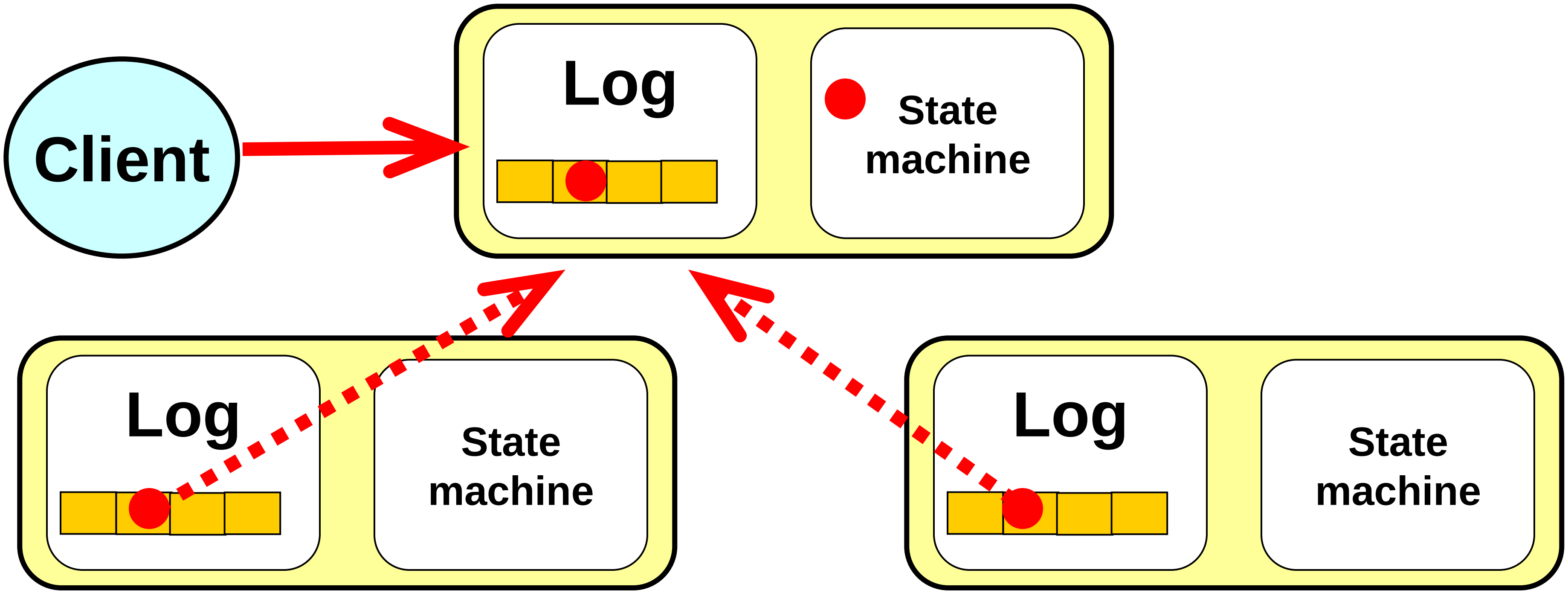
Leader stores request on its log and forwards it to its followers

The followers receive the request



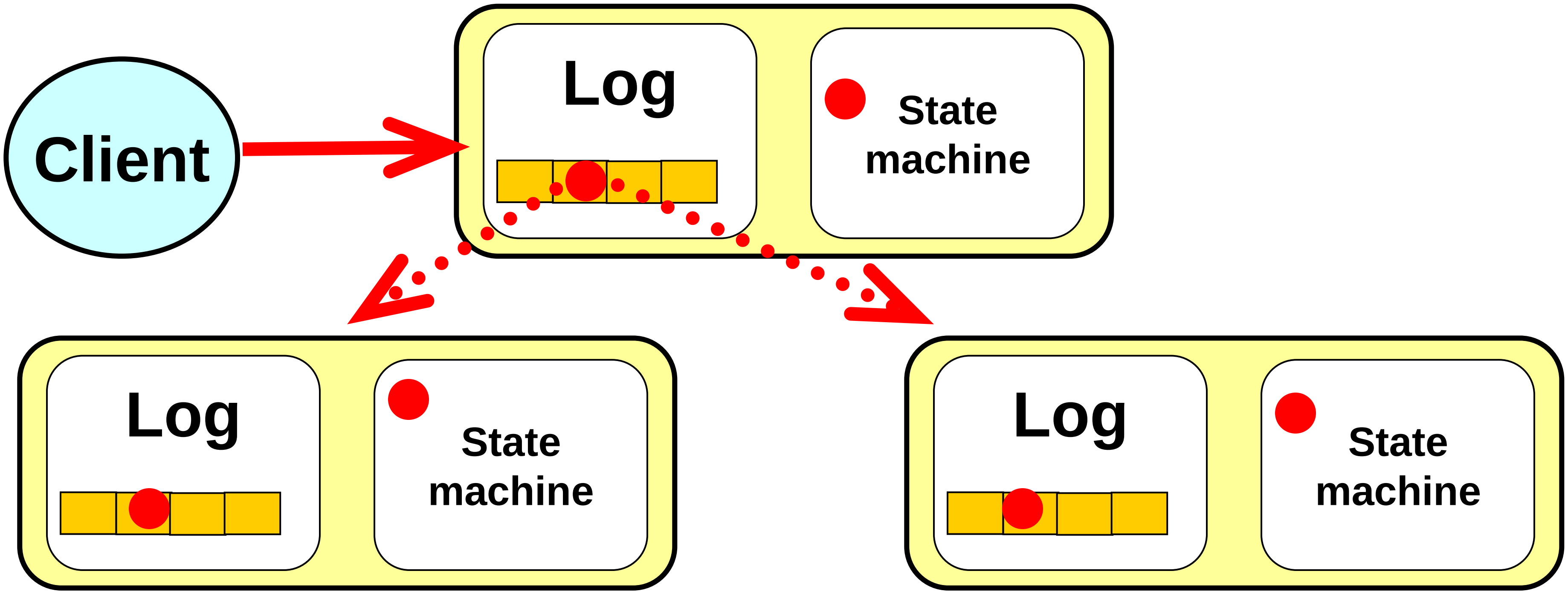
Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



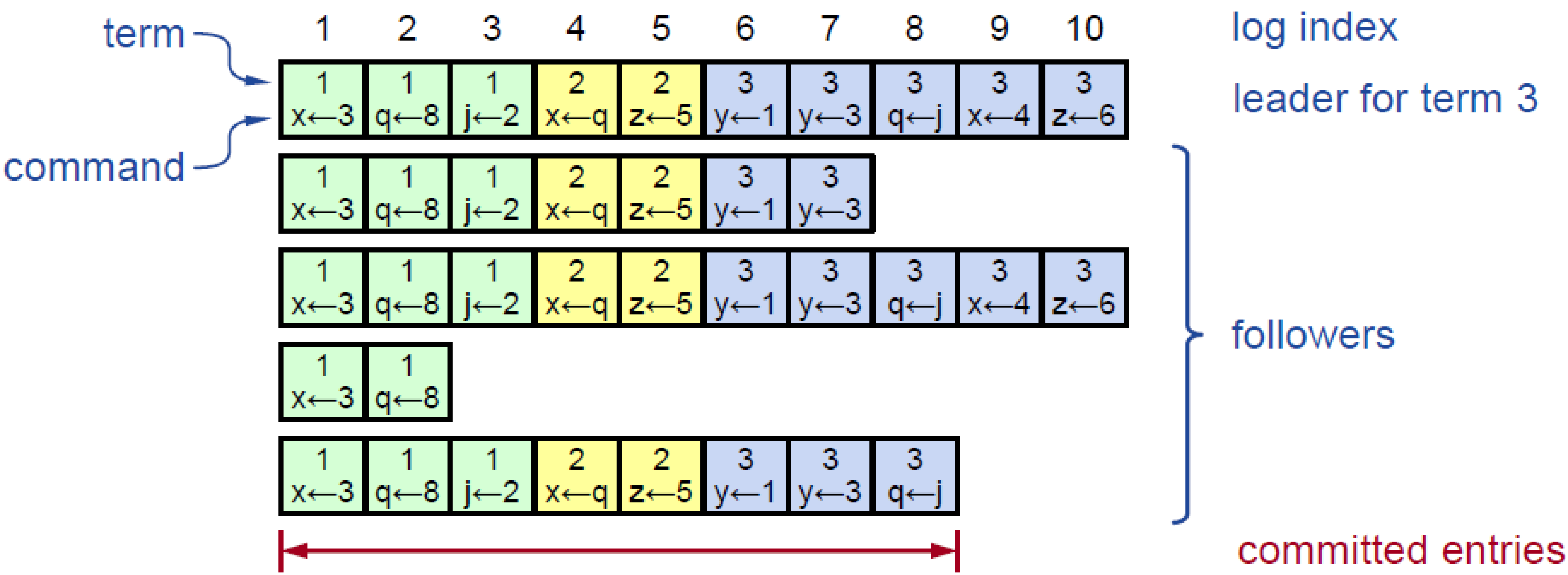
Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



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

Log structure



Must survive crashes (stored on disk)

Handling slow/crashed followers

- Leader retries AppendEntries RPCs until they succeed
  - They are idempotent

- Leader maintains a nextIndex for each follower
  - Index of entry it will send to that follower

Committed entries

- An entry is ***committed*** if it is replicated on a majority of servers by leader of its term

- A committed entry is safe to execute in state machines

Committed entries

- Guaranteed to be both
  - Durable
  - Eventually executed by all the available state machines
- Committing an entry also commits all previous entries
  - All AppendEntries RPCS - including heartbeats - include the index of its most recently committed entry

Committed entries

- 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

Safety

- Leader crash can result in log inconsistencies:

	1	2	3	4	5	6	7	8	9	10	log index
s_1	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3	3 q←j			leader for term 4
s_2	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3				followers
s_3	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3	3 q←j	3 x←4		followers
s_4	1 x←3	1 q←8									followers
s_5	1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	2 y←3	2 q←j	2 x←8	2 x←4		followers

When leader crashes

- The logs can be in an 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

Log inconsistencies

- Raft minimizes special code for repairing inconsistencies:
  - **Leader assumes its log is correct**
  - Normal operation will repair all inconsistencies
    - Rolling back AppendEntries calls is enough

Log matching

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

1	2	3	4	5	6	7	8	9	10
1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	3 y←3	3 q←j	3 x←4	3 z←6
1 x←3	1 q←8	1 j←2	2 x←q	2 z←5	3 y←1	4 x←z	4 y←7		

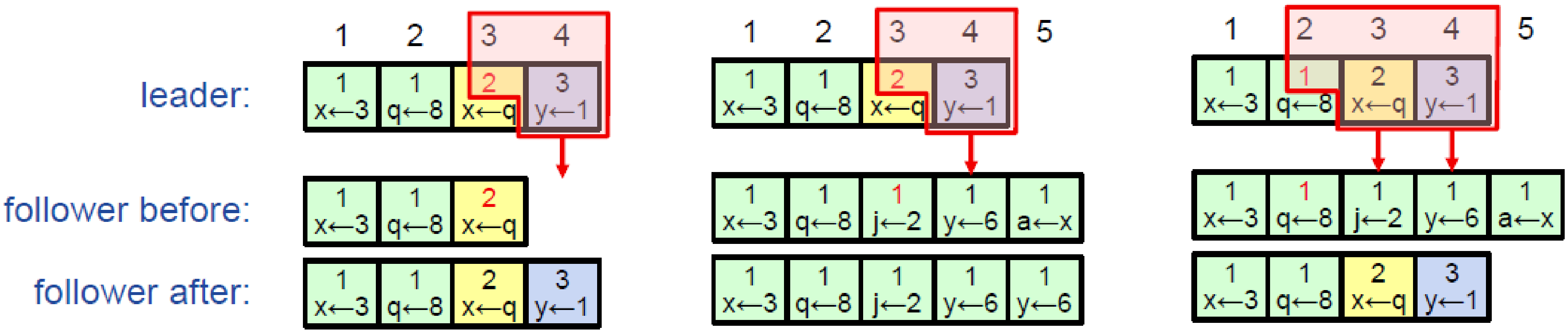
- If a given entry is committed, all preceding entries are also committed

AppendEntries consistency check

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

- Follower must contain matching entry; otherwise, it rejects request
  - Leader retries with lower log index

AppendEntries consistency check



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

- An induction step, ensures Log Matching Property

Leader completeness

- Once a log entry is committed, all future leaders must store that entry
- Must impose conditions on new leaders

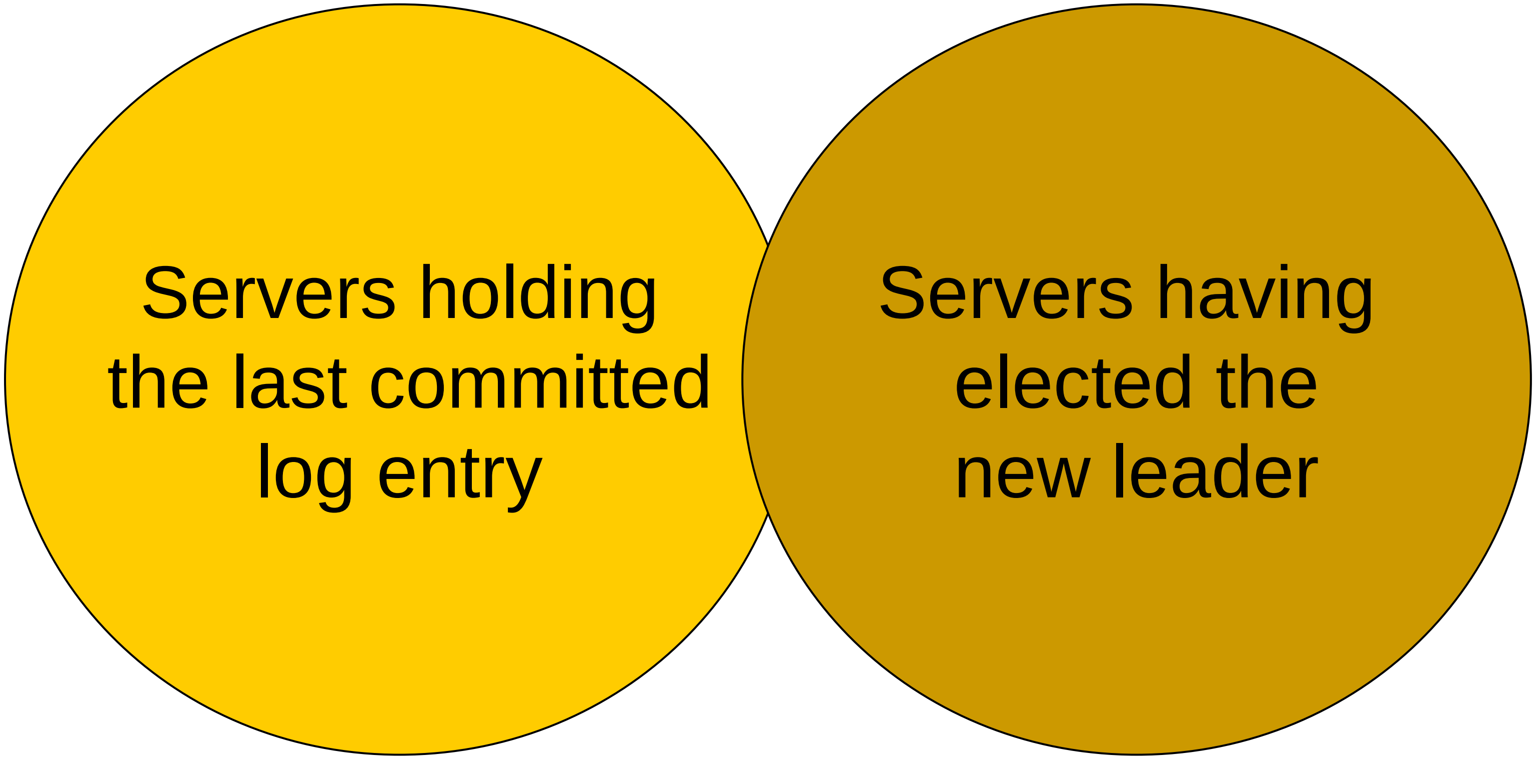
Leader completeness

- Servers with incomplete logs must not get elected:
  - Candidates include term and index of last log entry in RequestVote RPCs
  - Voting server denies vote if its log is more up-to-date
  - Logs ranked by <lastTerm, lastIndex>
- Majority rule does the rest

Leader election for term 4:

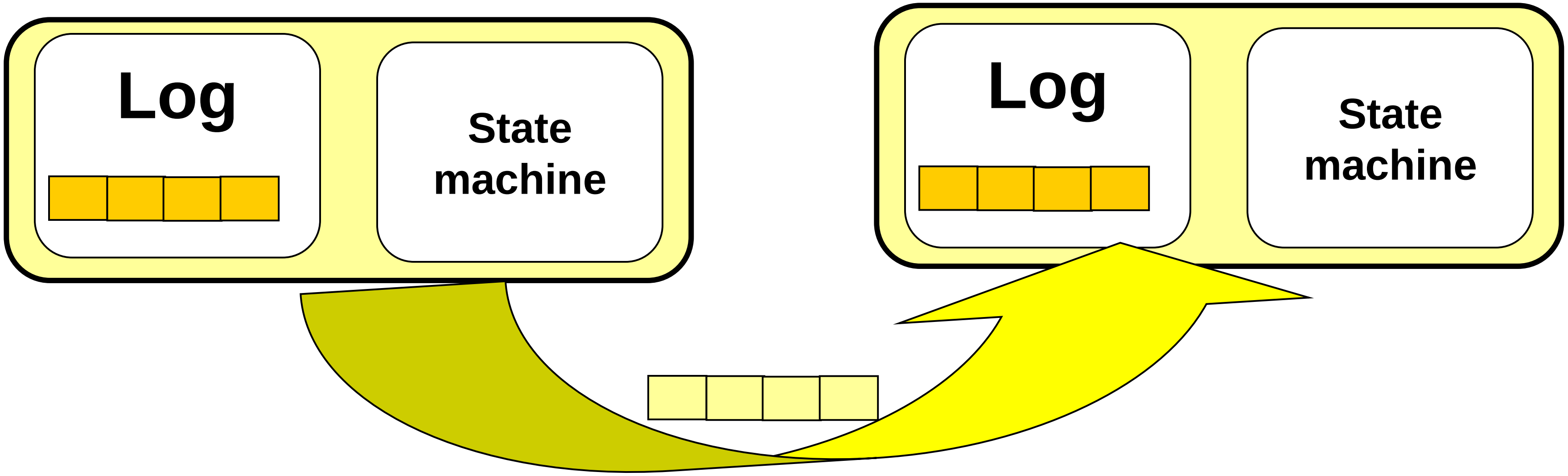
	1	2	3	4	5	6	7	8	9
s_1	1	1	1	2	2	3	3	3	
s_2	1	1	1	2	2	3	3		
s_3	1	1	1	2	2	3	3	3	3
s_4	1	1	1	2	2	3	3	3	
s_5	1	1	1	2	2	2	2	2	2

Leader completeness



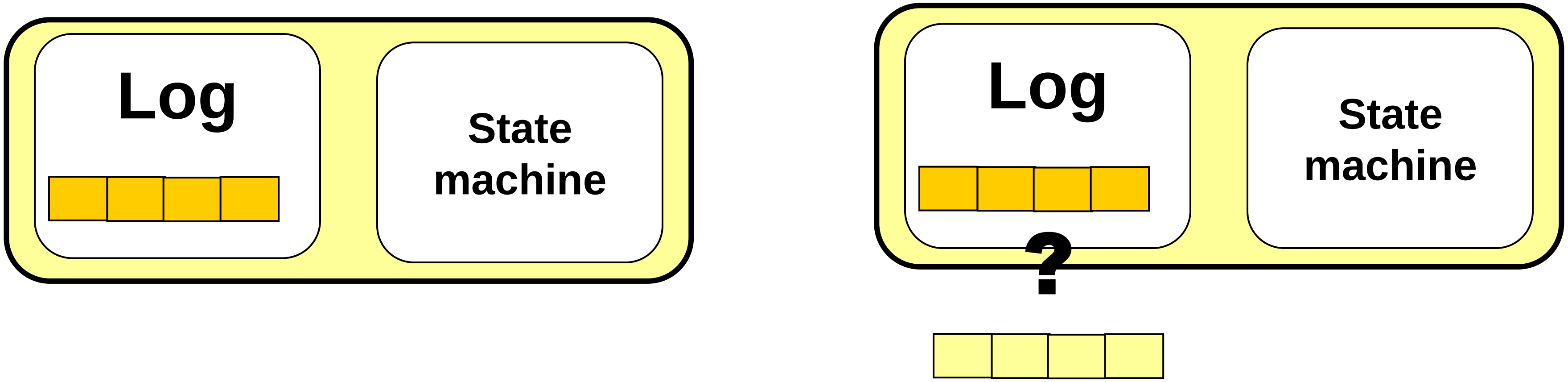
Two majorities must intersect

An election starts



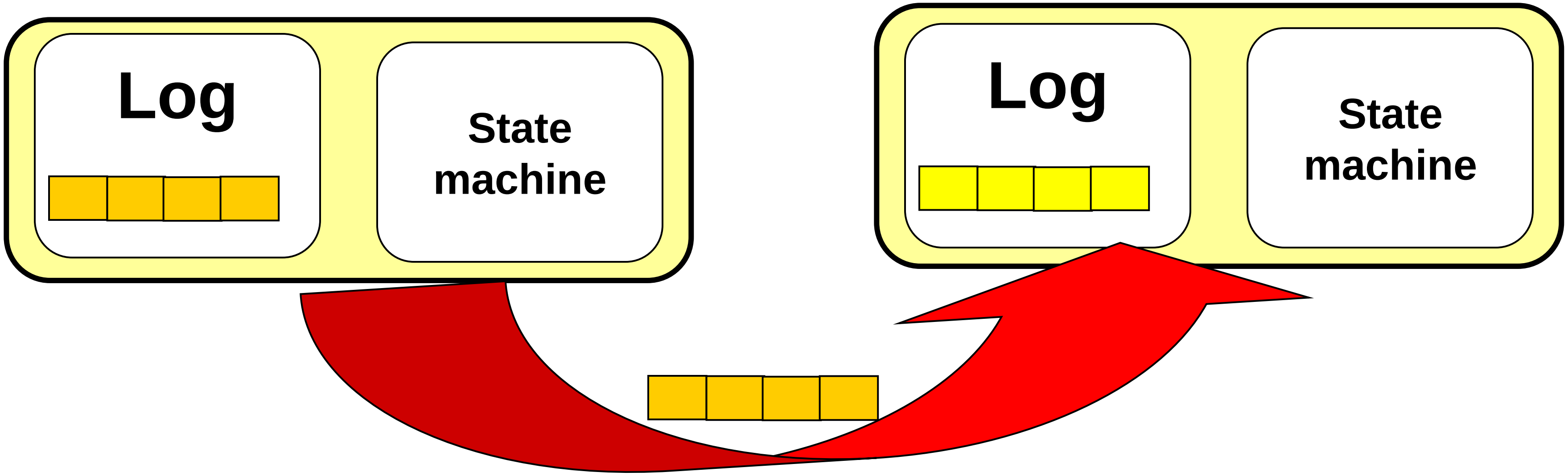
- Candidate for leader position requests votes of other former followers
  - Includes a summary of the state of its log

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

The new leader is in charge



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

The new leader is in charge

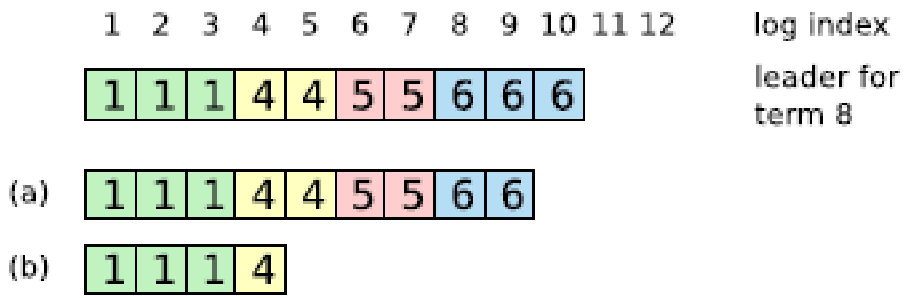
- New leader sets its nextIndex to the index just after its last log entry

- Broadcasts it to all its followers

The new leader is in charge

- Followers that have missed some AppendEntries calls will refuse all further AppendEntries calls
- Leader will decrement its nextIndex for that follower and redo the previous AppendEntries call
  - Process will be repeated until a point where the logs of the leader and the follower match
- Leader will then send to the follower all the log entries it missed

The new leader is in charge



- 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

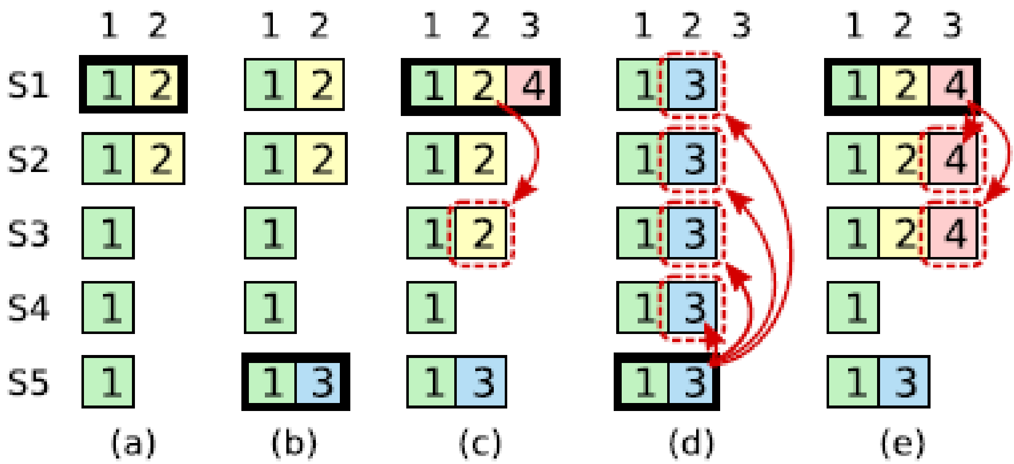
Committing entries

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

- Leader should only commit entries from the current term
- Once it has been able to do that for one entry, all prior log entries are committed indirectly
  - Any follower accepting an AppendEntries RPC implicitly acknowledges it has processed all previous AppendEntries RPCs

Committing entries

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



Committing entries

- 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.

Committing entries

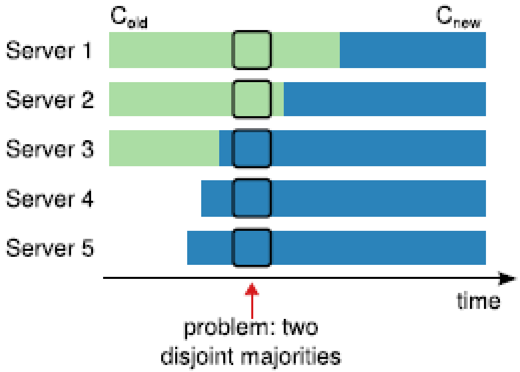
- If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3.
- However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win 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



Raft properties

- **Election Safety**: at most one leader can be elected in a given term
- **Leader Append-Only**: a leader never modifies or deletes entries in its log
- **Log Matching**: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index
- **Leader Completeness**: if a log entry is committed, then that entry will be present in the logs of all future leaders
- **State Machine Safety**: 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

Implementations

- Two thousand lines of C++ code, not including tests, comments, or blank lines.
- More than 80 independent third-party open-source implementations listed in Raft home page
- Some commercial implementations

Additional topics

- Understandability, correctness, performance
  - See paper
- Log compactation and client interaction (linearizability)
  - See TR (Ph.D. dissertation)

Conclusion

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

https://raft.github.io/

https://www.youtube.com/watch?v=vYp4LYbnnW8