Cloud-Based Data Processing

Consensus

Jana Giceva

Fault-tolerant total order broadcast

- Total order broadcast is very useful for state machine replication.
- Can implement total order broadcast by sending all messages via a single **leader.**

- **Problem: what if the leader crashes / becomes unavailable?**

- **Manual failover:**
  a human operator chooses a new leader, and reconfigures each node to use a new leader.

  Used in many databases. Fine for planned maintenance.
  Unplanned outage? Humans are slow, may take a long time until the system recovers.

- **Can we automatically choose a new leader?**

Consensus and total order broadcast

- Traditional formulation of consensus **several nodes come to an agreement about a single value.**

- In context of total order broadcast – this value is the next message to be delivered.
- Once one node **decides** on a certain message order, all nodes will decide the same order.

- A consensus algorithm must satisfy the following properties:
  - **Uniform agreement –** no two nodes decide differently
  - **Integrity –** no node decides twice
  - **Validity –** if a node decides value v, then v was proposed by some node.
  - **Termination –** every node that does not crash, eventually decides some value.

- **Common consensus algorithms:**
  - **Paxos (Lamport, '98):** single-value consensus
  - **Multi-Paxos:** generalization to total order broadcast
  - **Raft (Ongaro and Osterhaut'14), Viewstampted Replication, Zab:** FIFO-total order broadcast by default

Consensus system models

- Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

- Why not asynchronous?
  - **FLP result (**Fischer, Lynch, Paterson):
    There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
  - **Paxos, Raft, etc.** use clocks only used for timeouts/failure detector to ensure progress. Safety (correctness) does not depend on timing.

- There are also consensus algorithms for a partially synchronous **Byzantine** system model (used in Blockchain).

Core of consensus: Leader

- Leader election

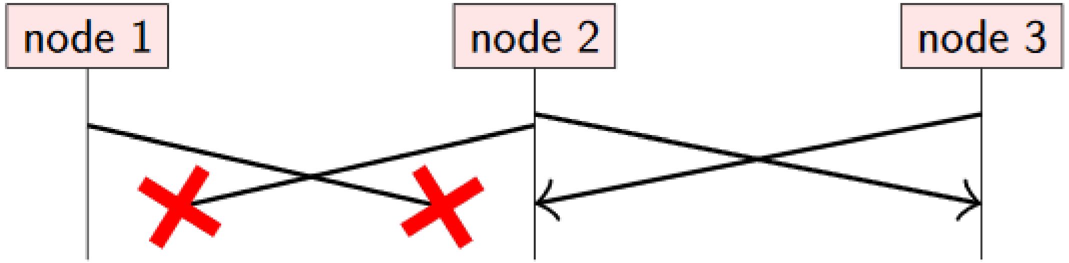
- Multi-Paxos, Raft, etc. use a leader to sequence messages.
  - Use a **failure detector** (timeout) to determine suspected crash or unavailability of a leader.
  - On suspected leader crash, **elect a new one.**
  - Prevent **two leaders at the same time** (“split brain” problem).

- Ensure <= 1 leader per **term:**
  - Term is incremented every time a leader election is started
  - A node can only **vote once per term**
  - Require a **quorum** of nodes to elect a leader in a term

Can we guarantee there is only one leader?

- Can guarantee unique leader **per term.**
- **Cannot** prevent having multiple leaders from different terms.

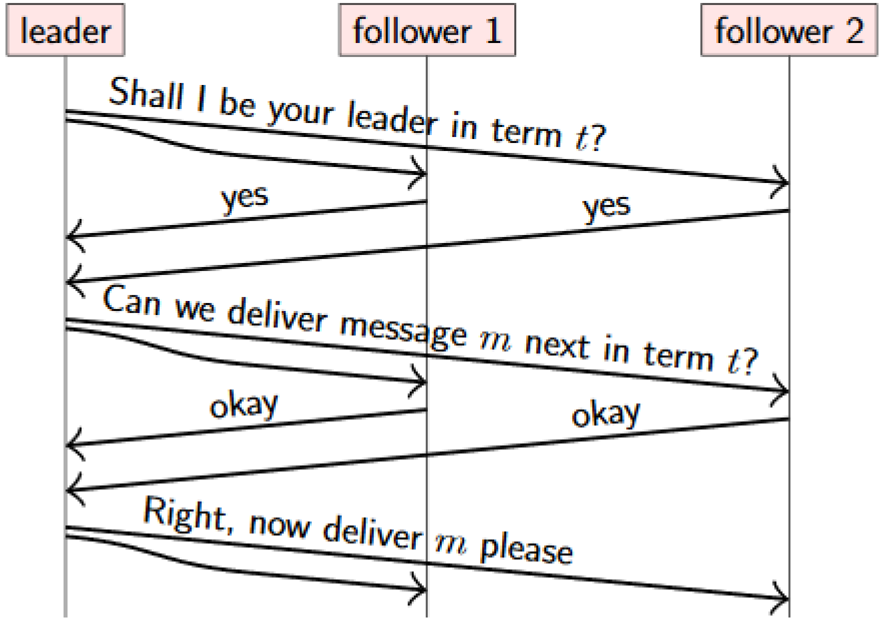
Example: node 1 is leader in term t, but due to network partitioning, it can no longer communicate with nodes 2 and 3.



Nodes 2 and 3 may elect a new leader in term $t + 1$.
Node 1 may not even know that a new leader has been elected!

Checking if a leader has been voted out.

- For every decision (message to deliver), the leader must first get acknowledgement from a quorum.


leader
follower 1
follower 2
Shall I be your leader in term t?
yes
yes
Can we deliver message m next in term t?
okay
okay
Right, now deliver m please

The Raft consensus algorithm

Raft Decomposition

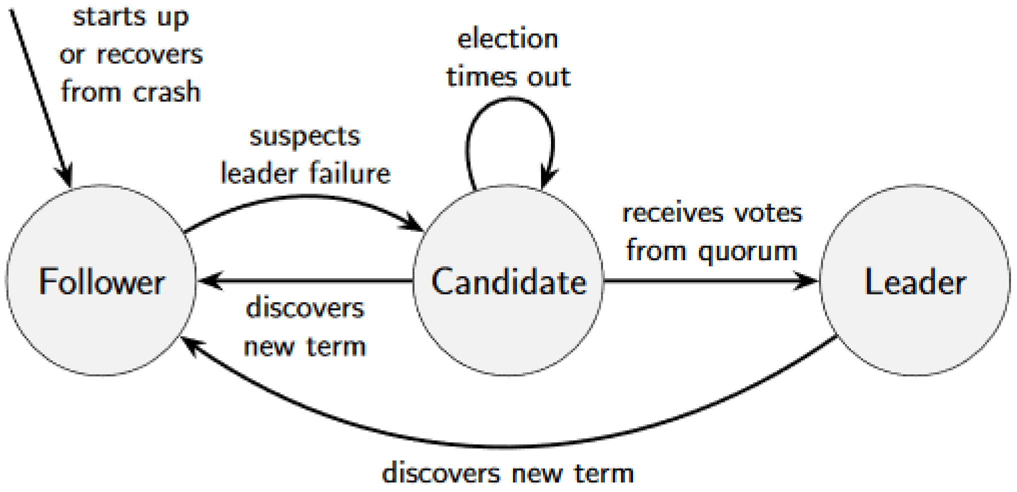
1. **Leader election:**
   - Select one server to act as leader
   - Detect crashes, choose new leader

2. **Normal operation: log replication**
   - Leader accepts commands from clients, appends to its log
   - Leader replicates its log to other servers (overwrites inconsistencies)

Raft Overview

- **Leader election**
  - Select one of the servers to act as a leader
  - Detect crashes, choose new leader
- **Normal operation (basic log replication)**
- **Safety and consistency after leader changes**
- **Neutralizing old leaders**
- **Client interactions**
  - Implementing linearizable semantics
- **Configuration changes**
  - Adding and removing servers

Node states and transitions in Raft



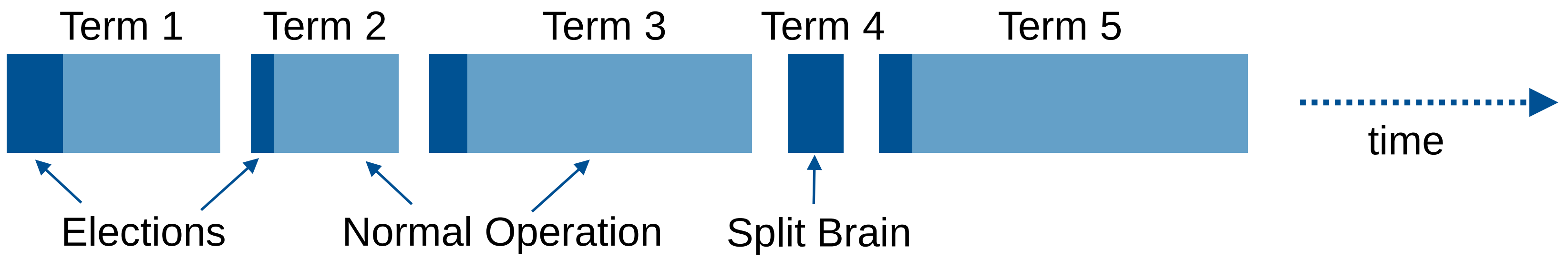
- **Normal operation:** 1 Leader, N-1 followers.

- **Follower**
  passive, but expects regular heartbeats

- **Candidate**
  active, issues **RequestVote** RPCs to get elected as a leader

- **Leader**
  active, issues **AppendEntries** RPCs
  – Replicates its log
  – Heartbeats to maintain leadership

Terms



- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value (no global view)**
  - 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**

Raft Protocol Summary

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

RequestVote 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

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return 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
8. Advance state machine with newly committed entries

Heartbeats and Timeouts

- **Servers start up as Followers**

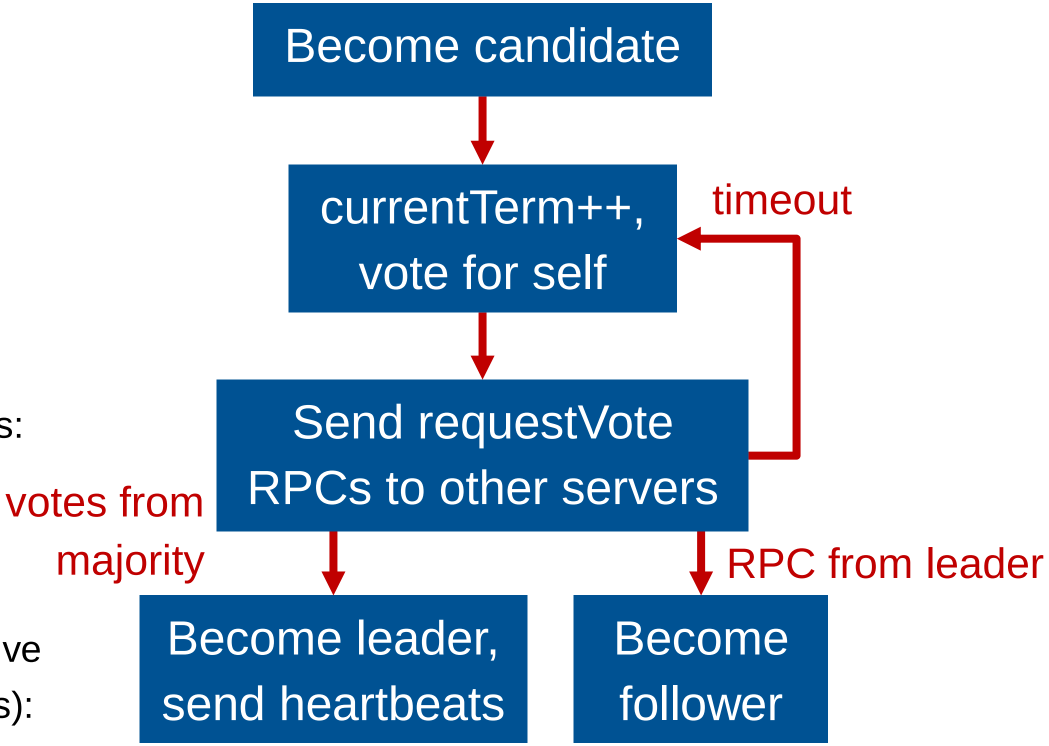
- **Followers expect to receive RPCs from leaders or candidates**

- Leaders must send **heartbeats** (empty AppendEntry RPCs) to maintain authority

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

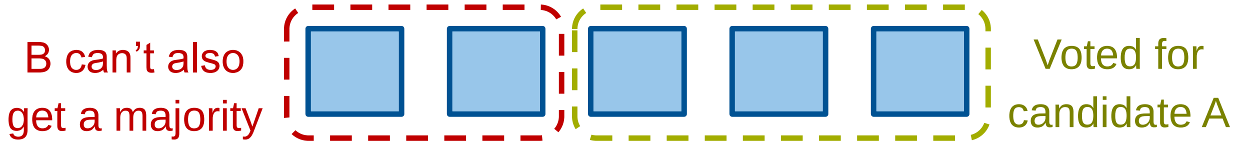
Election basics

- **Increment current term**
- **Change to Candidate state**
- **Vote for self**
- **Send RequestVote RPCs to all other servers, retry until either:**
  1. Receive votes from majority of servers:
     – become a leader,
     – send heartbeats
  2. Receive RPC from valid leader:
     – return to follower state, become passive
  3. No-one wins election (timeout elapses):
     – Increment term, start new election


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

Election correctness

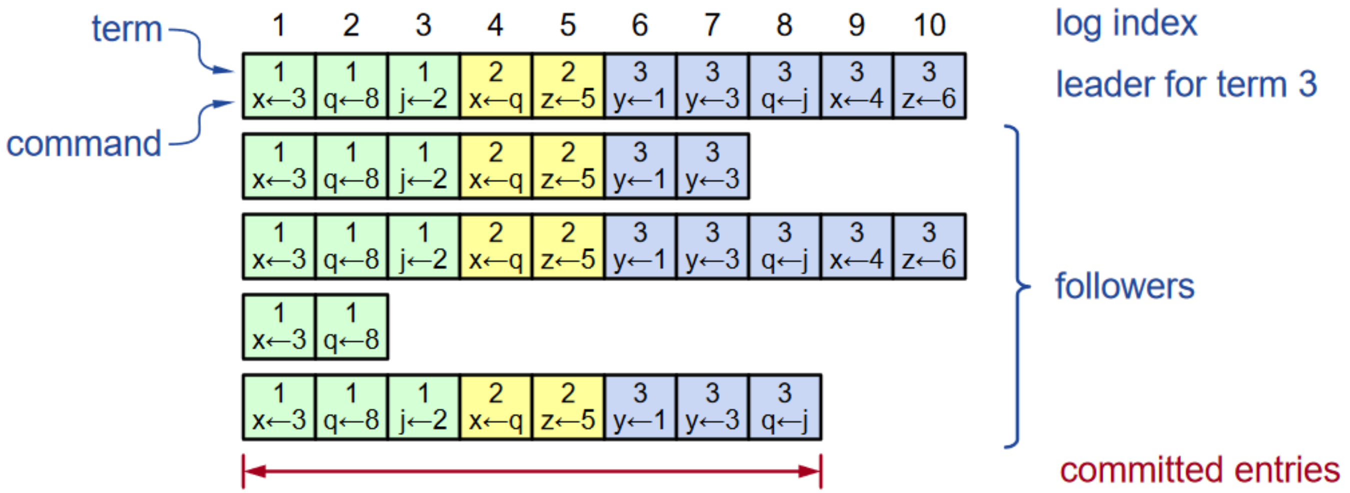
- **Safety: allow at most one winner per term**
  - Each server gives only **one vote** per term (persist on disk)
  - Majority is required to win election



- **Liveness: some candidate must eventually win**
  - Choose election timeouts randomly in [T, 2T] (e.g., 150-300ms)
  - One server usually times out and wins election before other time out
  - Works well if T >> broadcast time

- **Randomized approach simpler than ranking**

Log Structure



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

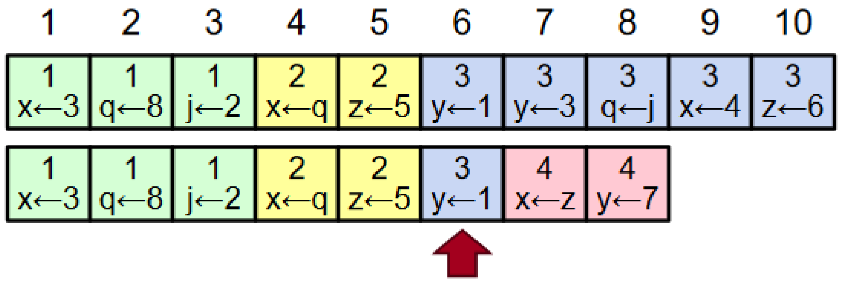
- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**
- **Entry committed if safe to execute in state machines**
  - Replicated on **majority** of servers **by leader of its term**

Normal Operation

- **Client sends command to leader**
- **Leader appends command to its log**
- **Leader sends AppenEntries RPCs to all followers**
- **Once new entry committed**:
  - 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
- **Crashed/slow followers?**
  - Leader retries AppendEntries RPCs until they succeed
- **Optimal performance in common case:**
  - One successful RPC to any majority of servers

Log Consistency

- **Goal: high level of consistency between the logs**
  - **If log entries on different servers have the same index and term**
    - They store the same command
    - The logs are identical in all preceding entries

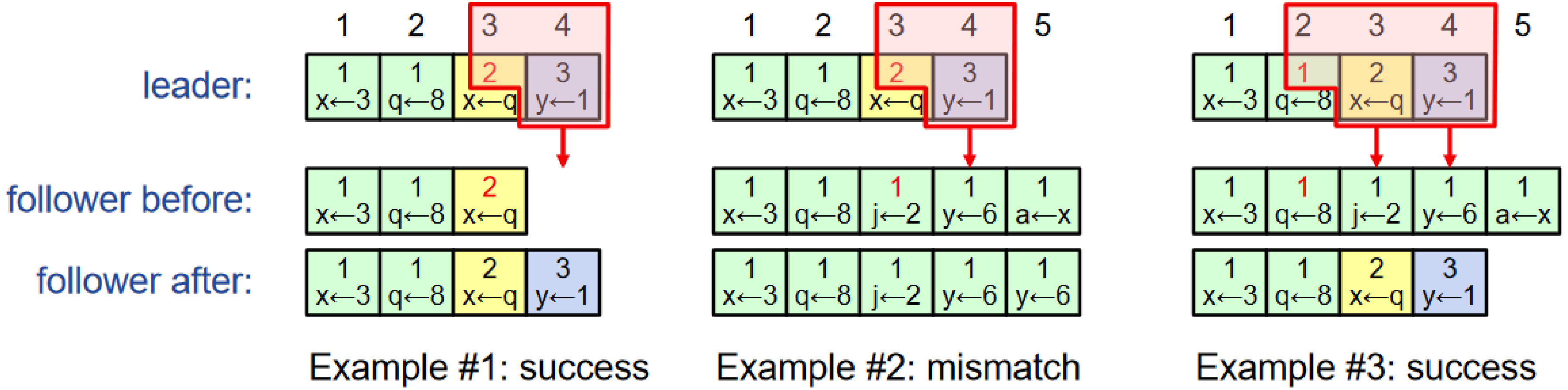


1 2 3 4 5 6 7 8 9 10
1 x←3 | 1 q←8 | 1 j←2 | 2 x←q | 2 z←5 | 3 y←1 | 3 y←3 | 3 q←j | 3 x←4 | 3 z←6
1 x←3 | 1 q←8 | 1 j←2 | 2 x←q | 2 z←5 | 3 y←1 | 4 x←z | 4 y←7

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

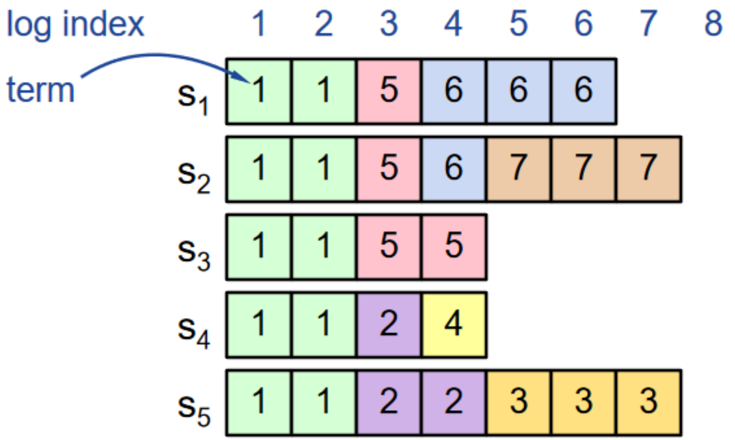
AppendEntries Consistency Check

- **AppendEntries RPCs include <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**



Leader Changes

- At the beginning of a new leader's term:
  - Old leader may have left entries partially replicated
  - No special steps by new leader: just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous log entries:



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

Safety Requirement

- **Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry**

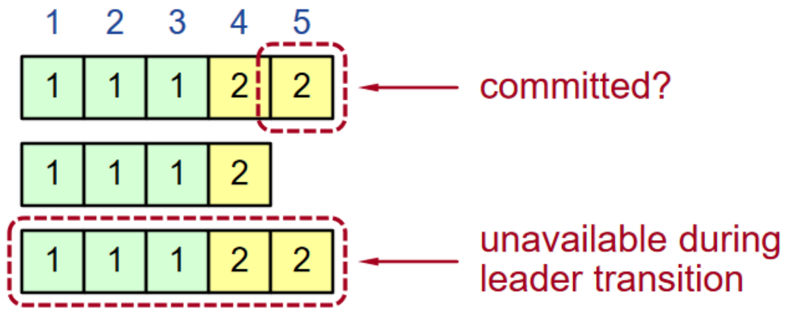
- **Raft safety protocol**
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders

- **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

- Committed -> Present in future leader's logs
  - Restrictions on commitment vs. restrictions on leader election

Picking the Best Leader

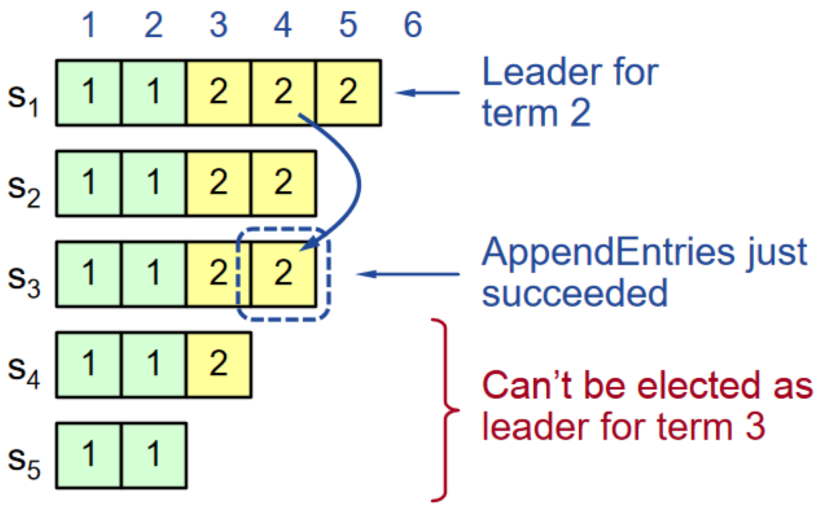
- **Can’t tell which entries are committed!**



- **During elections, choose candidate with log most likely to contain all committed entries.**
  - Candidates include index and term of last log entry in RequestVote RPCs
  - Voting server denies vote if its log is more up-to-date
  - Logs ranked by <lastTerm, lastIndex>

Committing Entry from Current Term

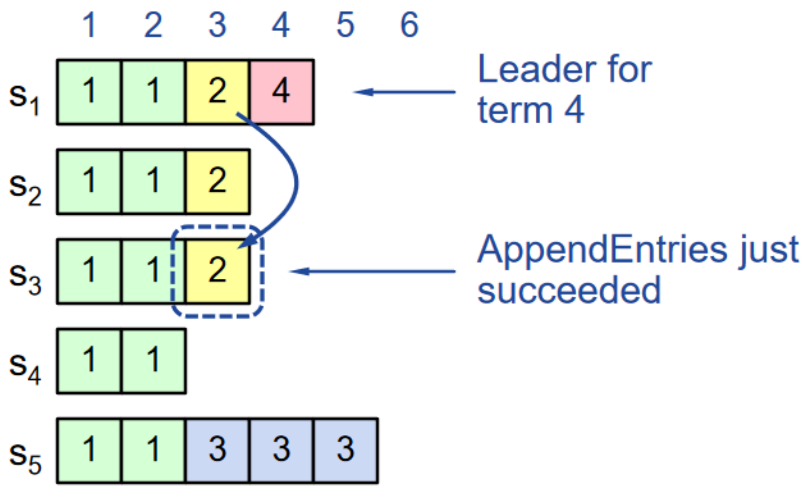
- **Case #1/2: Leader decides entry in current term is committed**



- **Safe: leader for term 3 must contain entry 4**

Committing Entry from Earlier Term

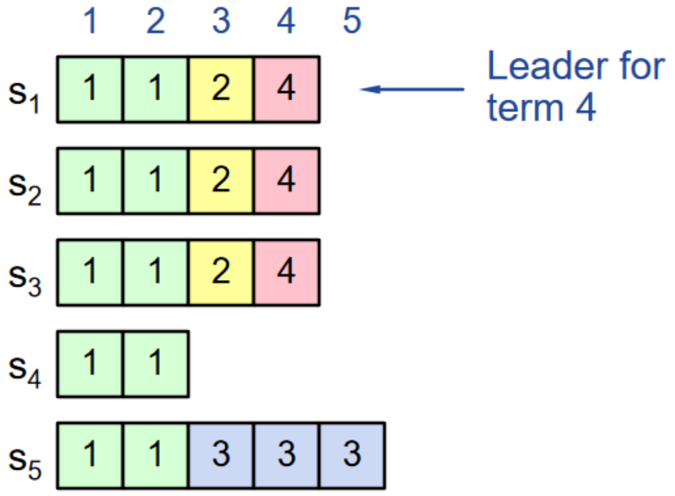
- **Case #2/2: Leader is trying to finish committing entry from an earlier term**



- **Entry 3 not safely committed:**
  - S_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on S_1, S_2 and S_3!

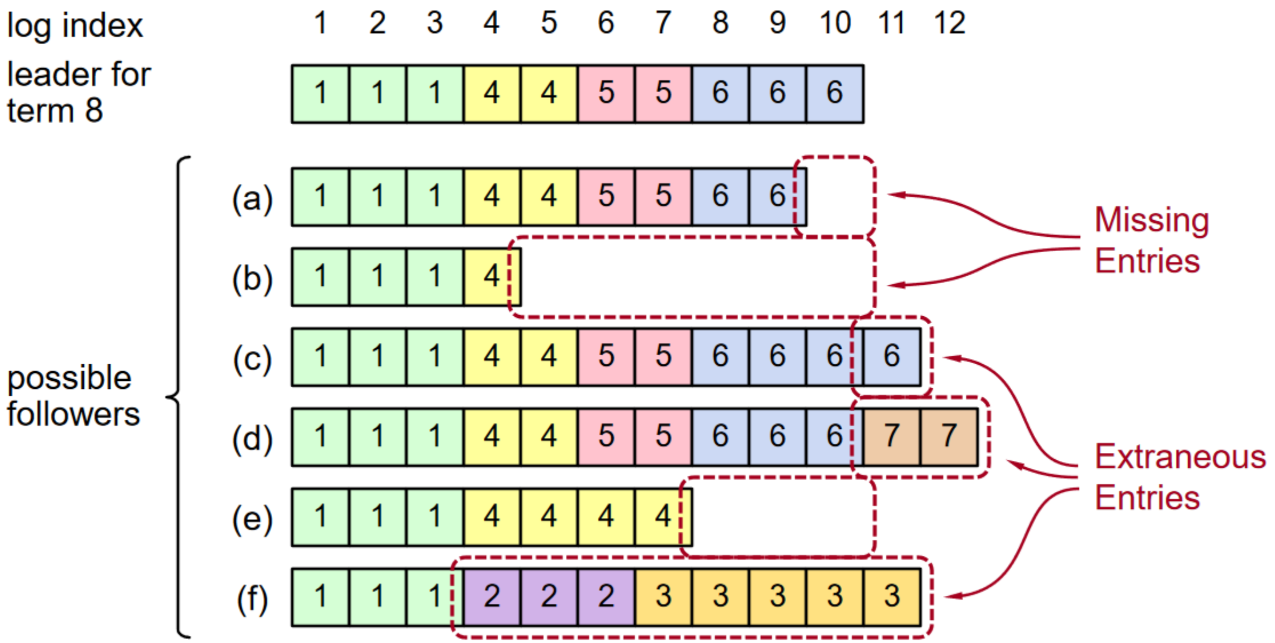
New Commitment Rules

- **For a leader to decide an 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
- **Once entry 4 is committed:**
  - S_5 cannot be elected leader for term 5
  - Entries 3 and 4 are both safe.
- **Combination of election and commitment rules makes Raft safe**



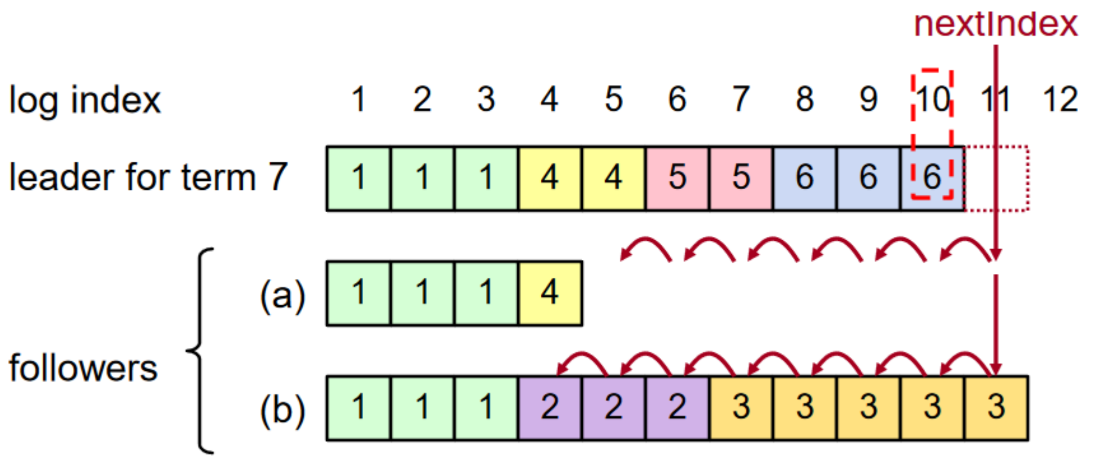
Log Inconsistencies

- **Leader changes can result in log inconsistencies**



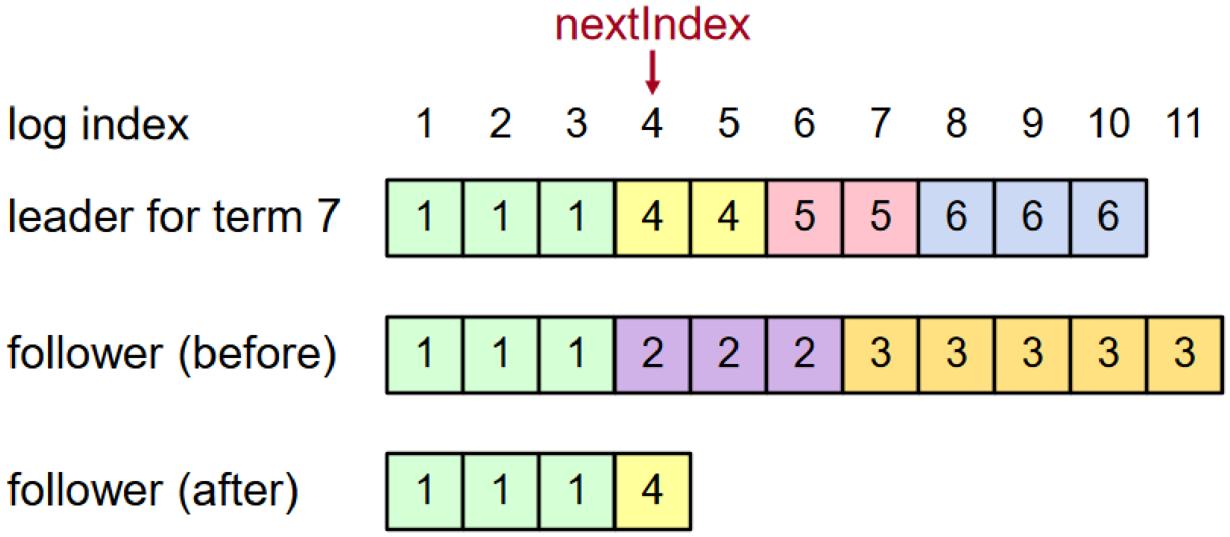
Repairing Follower Logs

- **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)
- **When AppendEntries consistency check fails, decrement nextIndex and try again:**



Repairing Logs, continued

- **When follower overwrites inconsistent entry, it deletes all subsequent entries:**



Neutralizing Old Leaders

- **Deposed leader may not be dead:**
  - Temporarily disconnected from the 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 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

- **Election updates the terms of majority of servers**
  - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader

- **Leader does not respond until command has been logged, committed, and executed on the leader’s state machine**

- **If a request times out (e.g., leader crash):**
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol, continued

- **What if a leader crashes after executing command, but before responding?**
  - Must not execute the command twice.

- **Solution: client embeds a unique id in each command**
  - Server includes id in log entry
  - Before accepting command, the leader checks its log for entry with that id
  - If id found in log, ignore the new command, return response from old command

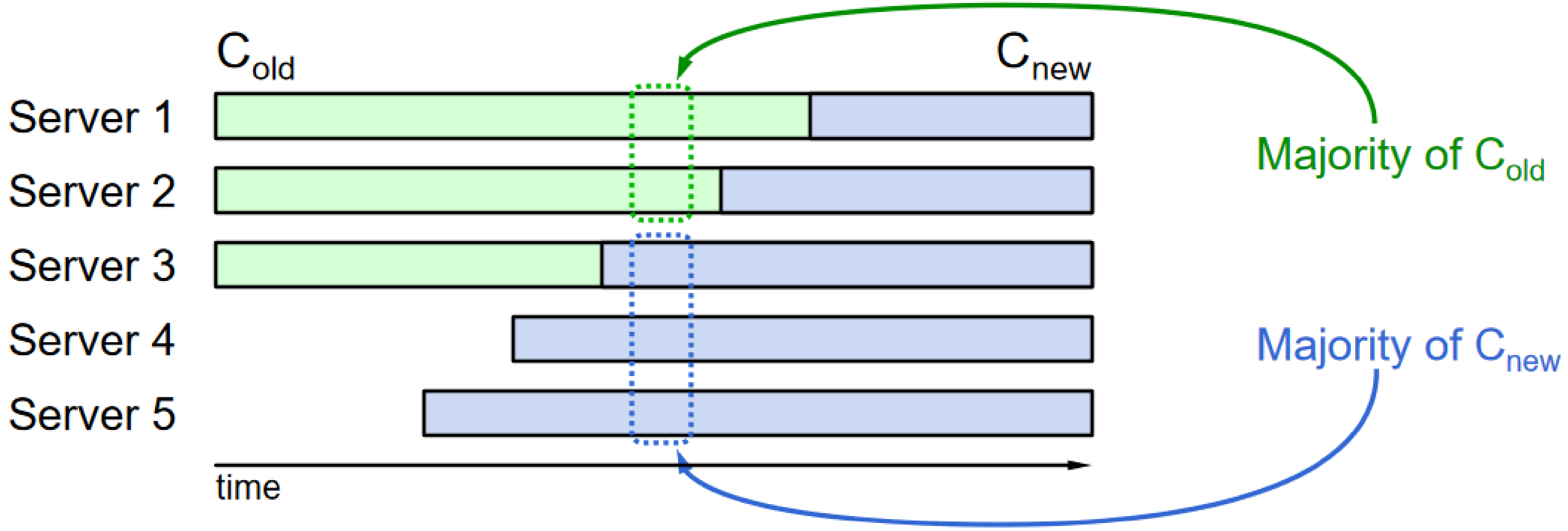
- **Result exactly-once semantics as long as client does not crash**

Configuration Changes

- **System configuration:**
  - ID, address for each server
  - Determines what constitutes a majority
- **Consensus mechanism must support changes in the configuration:**
  - Replace a failed machine
  - Change degree of replication

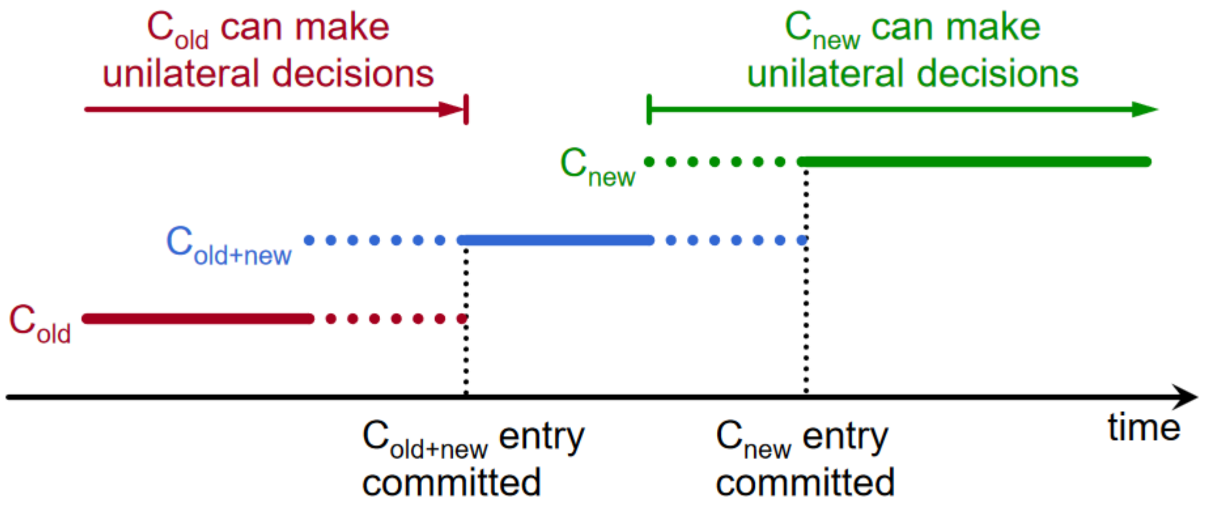
Configuration Changes, continued

- **Cannot switch directly from one configuration to another: conflicting majorities may arise**



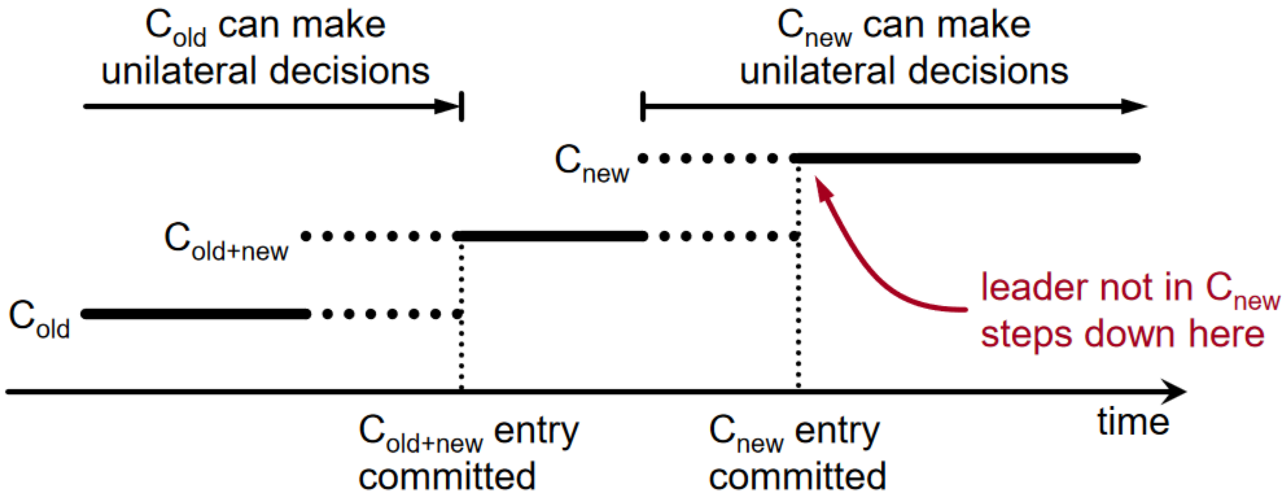
Joint Consensus

- **Raft uses a 2-phase approach:**
  - Intermediate phase uses **joint consensus**
    (i.e., needs majority of both old and new configuration for elections, committed).
  - Configuration change is just a log entry; applied immediately on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration


C_{old} can make unilateral decisions
C_{new} can make unilateral decisions
C_{new}
$C_{old+new}$
C_{old}
$C_{old+new}$ entry committed
C_{new} entry committed
time

Joint Consensus, continued

- **Additional details:**
  - Any server from either configuration can serve as leader
  - If current leader is not in C_{new}, it must step down once C_{new} is committed



C_{old} can make unilateral decisions
C_{new} can make unilateral decisions
C_{new}
$C_{old+new}$
C_{old}
leader not in C_{new} steps down here
time
$C_{old+new}$ entry committed
C_{new} entry committed

Raft Summary

1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes

Graphical visualization of the Raft protocol

- http://thesecretlivesofdata.com/raft/

Reference for paper and pseudo-code

- https://raft.github.io/

Limitations of consensus

- Consensus brings a list of safety properties to systems where everything else is uncertain:
  - Support for **agreement, integrity** and **validity**, and **fault-tolerant**!

- But that all comes at a cost:
  - **Synchronous-based replication**
    - Much worse performance than asynchronous
  - **Strict quorum majority to operate**
    - Needs a minimum of 3 nodes to tolerate 1 failure, or minimum of 5 nodes to tolerate 2 failures
  - **Static membership algorithm**
    - Cannot simply add or remove nodes in the cluster
  - **Relies on timeouts to detect failed nodes**
    - Known to have issues for highly variable network delays

References

The material covered in this class is mainly based on:

- The Raft lecture slides from John Ousterhaut and Diego Ongaro (Stanford) (link)
- The book *“Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”* by Martin Kleppmann (Chapter 9) (link)
- Slides from “*Distributed Systems”* course from University of Cambridge (link)
- Raft (https://raft.github.io/)