The Raft Consensus Algorithm

Diego Ongaro and John Ousterhout

Stanford University

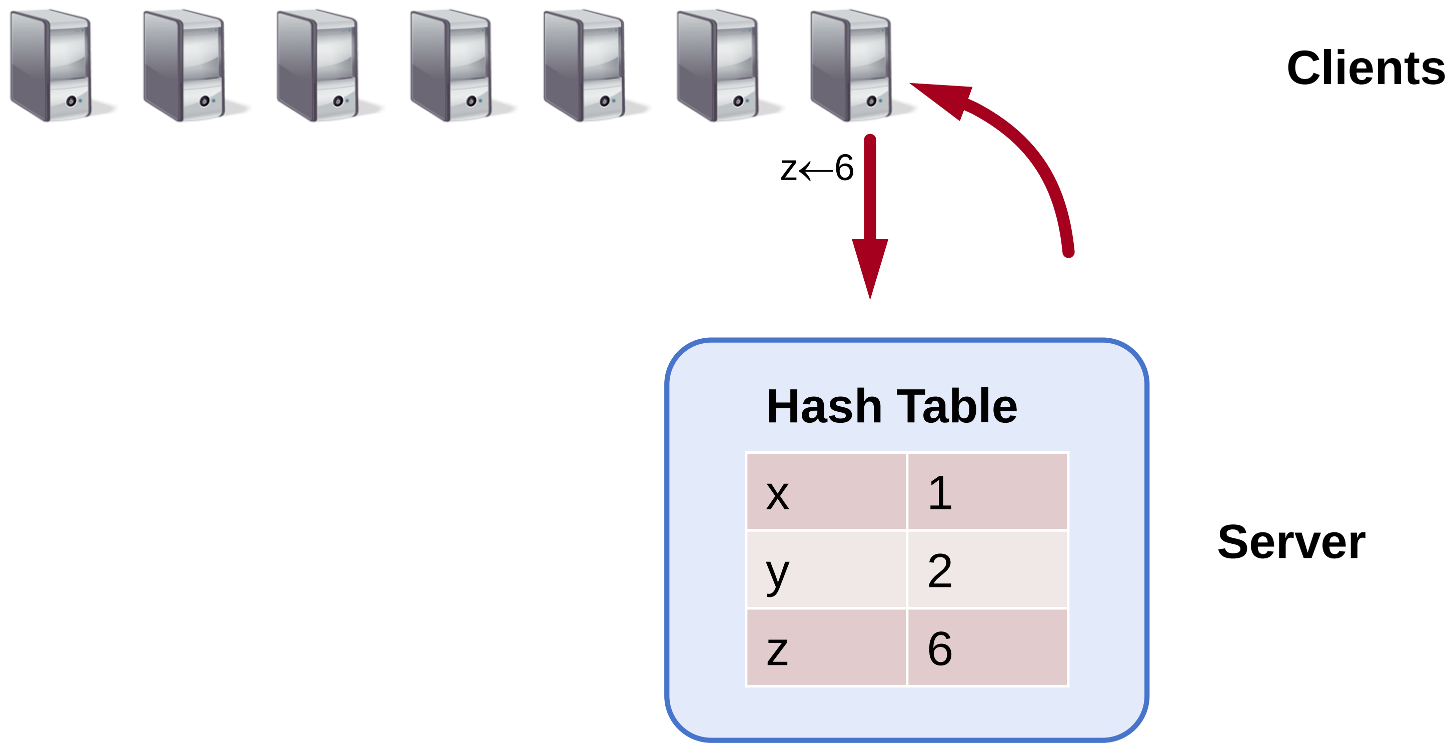
What is Consensus?

- **Consensus: get multiple servers to agree on state**
- **Solutions typically handle minority of servers failing**
- **== master-slave replication that can recover from master failures safely and autonomously**
- **Used in building consistent storage systems**
  - Top-level system configuration
  - Sometimes manages entire database state (e.g., Spanner)
- **Examples: Chubby, ZooKeeper, Doozer**

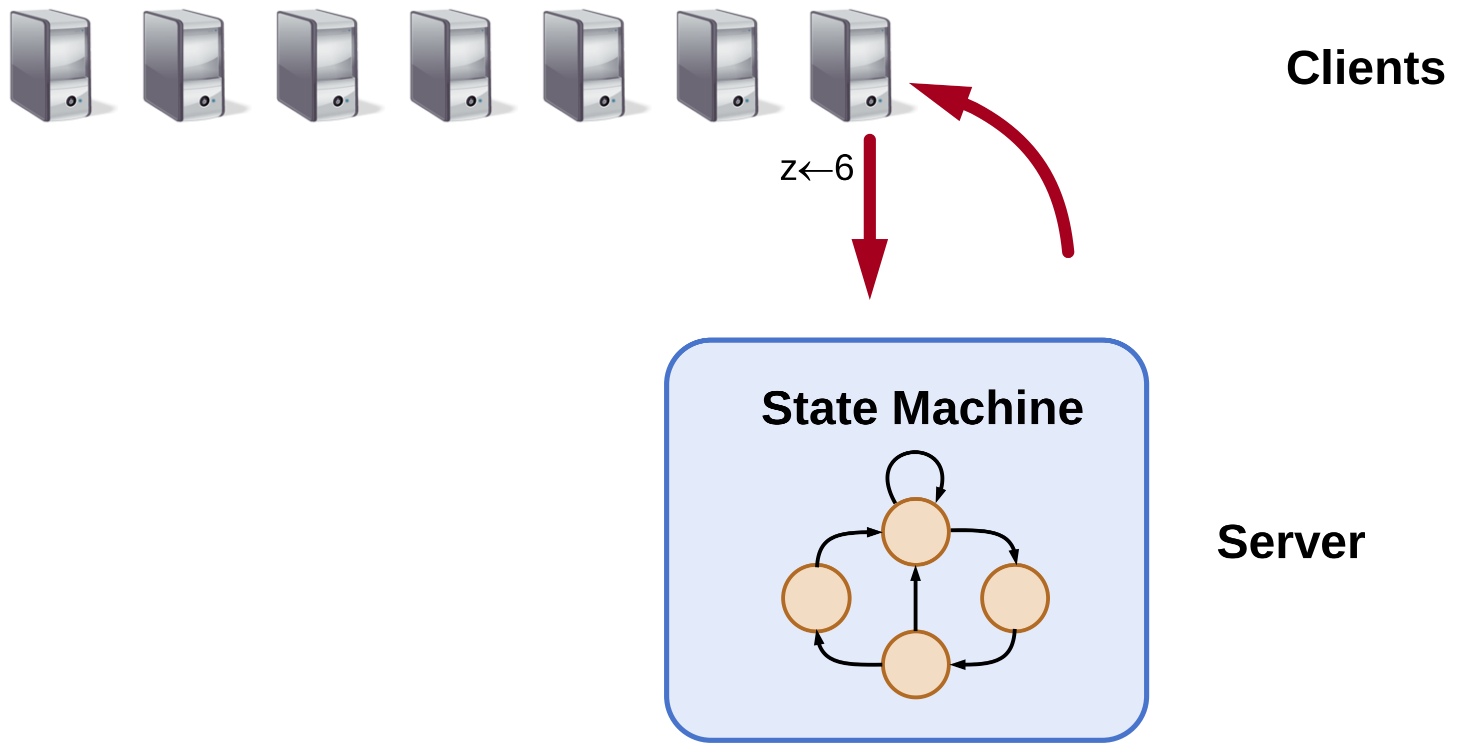
Raft: making consensus easier

- **Consensus widely regarded as difficult**
  - Dominated by an algorithm called Paxos
- **Raft designed to be easier to understand**
  - User study showed students learn Raft better
- **25+ implementations of Raft in progress on GitHub**
  - See http://raftconsensus.github.io
  - Bloom, C#, C++, Clojure, Elixir, Erlang, F#, Go, Haskell, Java, Javascript, OCaml, Python, Ruby

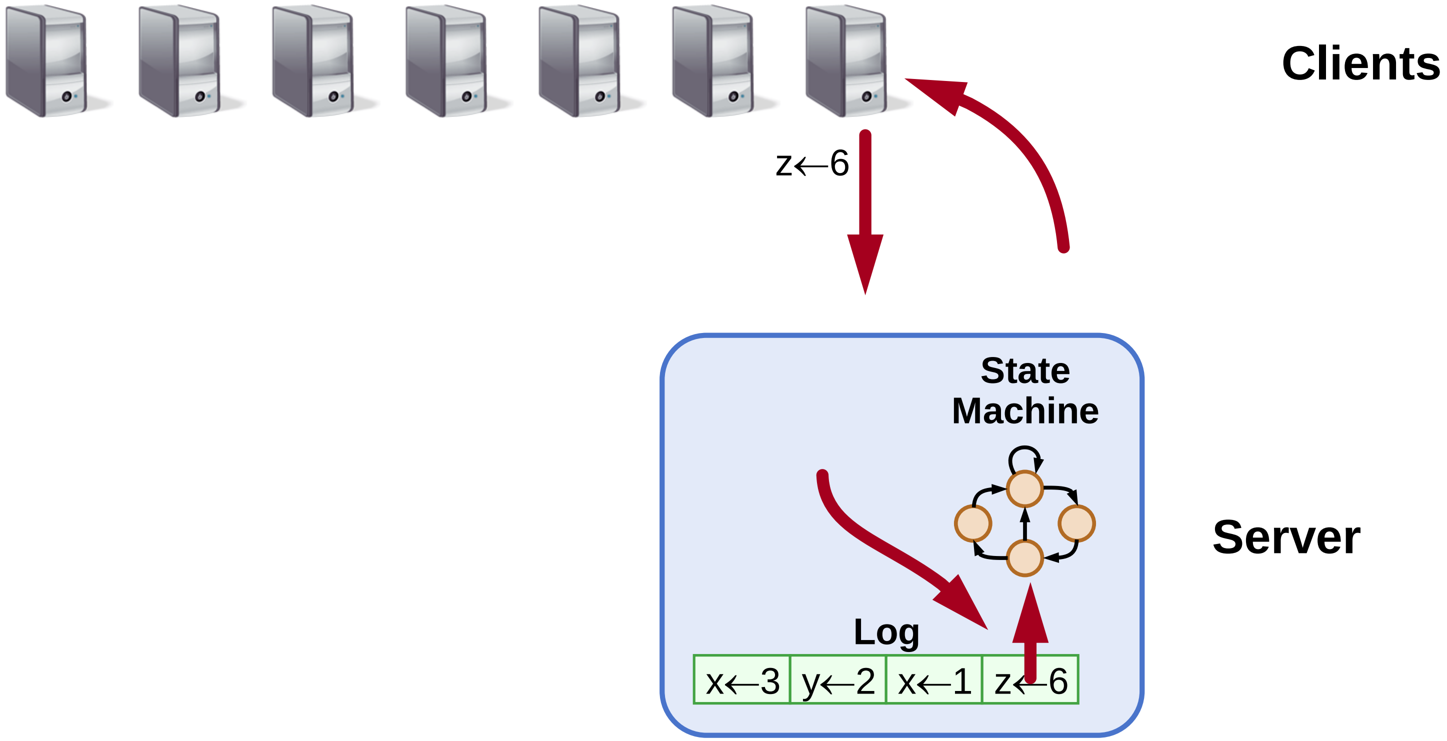
Single Server



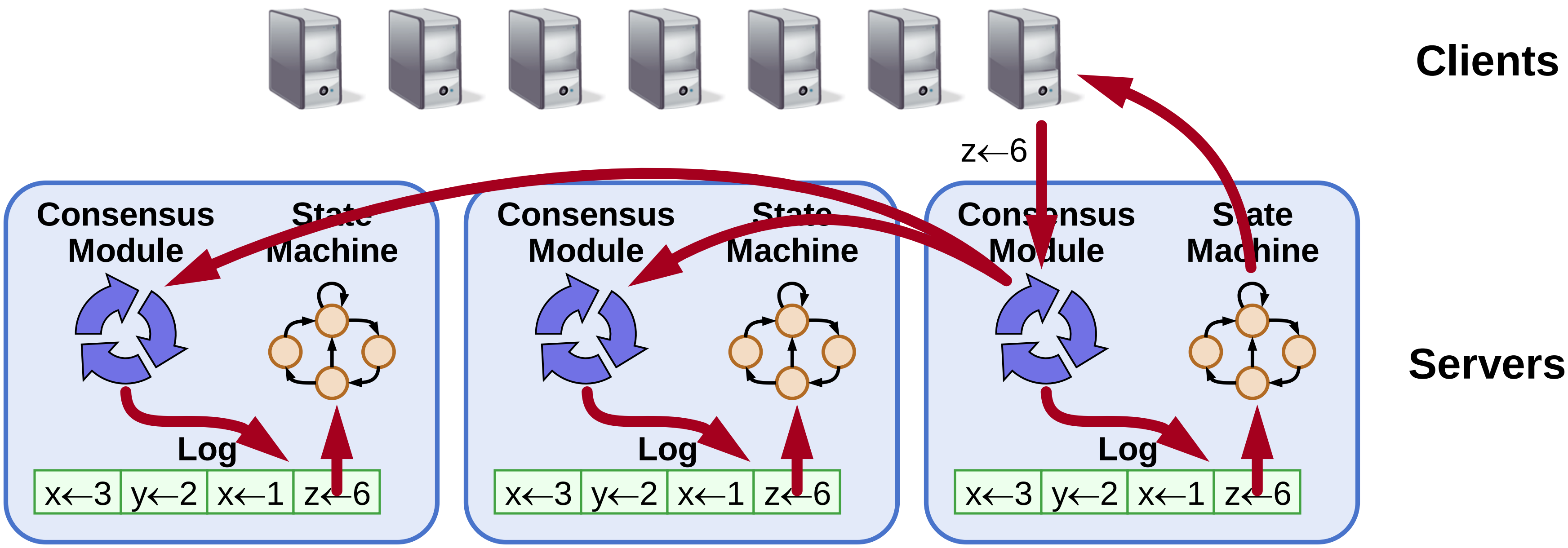
Single Server



Single Server



Goal: Replicated Log



- **Replicated log ⇒ replicated state machine**
  - All servers execute same commands in same order
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

Approaches to Consensus

Two general approaches to consensus:

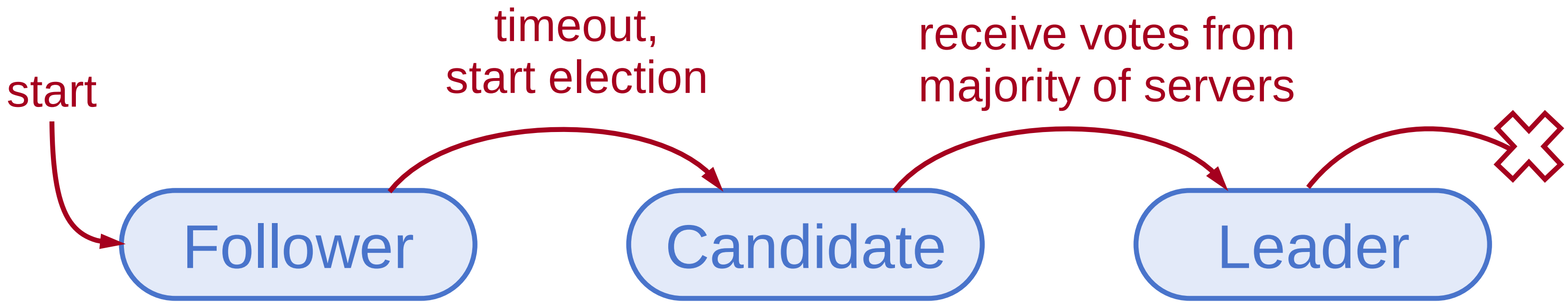
- **Symmetric, leader-less:**
  - All servers have equal roles
  - Clients can contact any server
- **Asymmetric, leader-based:**
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
- **Raft uses a leader:**
  - Decomposes the problem (normal operation, leader changes)
  - Simplifies normal operation (no conflicts)
  - More efficient than leader-less approaches

Raft Overview

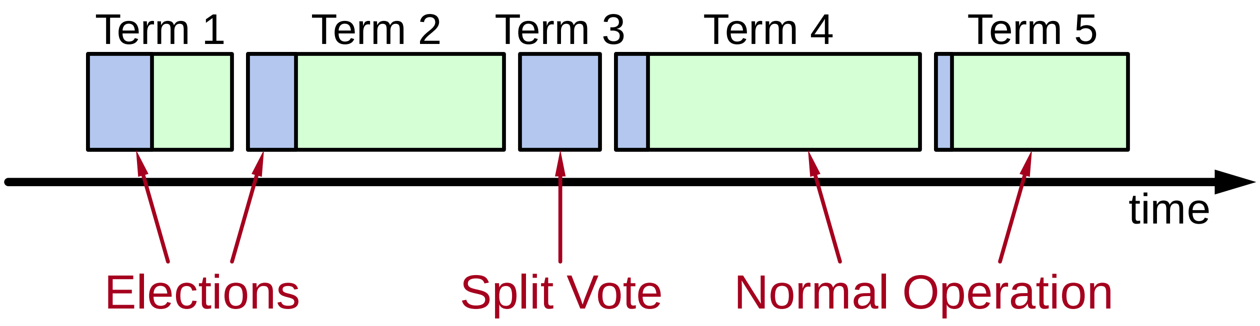
1. **Leader election**:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. **Normal operation (log replication)**
   - Leader takes commands from clients, appends them to its log
   - Leader replicates its log to other servers (overwriting inconsistencies)
3. **Safety**
   - Need committed entries to survive across leader changes
   - Define commitment rule, rig leader election

Server States

- **At any given time, each server is either:**
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive replica (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader



Terms



- **Time divided into terms:**
  - Election
  - Normal operation under a single leader
- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value**
- **Key role of terms: identify obsolete information**

Heartbeats and Timeouts

- **Servers start up as followers**
- **Followers expect to receive RPCs from leaders or candidates**
- **If election timeout elapses with no RPCs:**
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms
- **Leaders must send heartbeats to maintain authority**

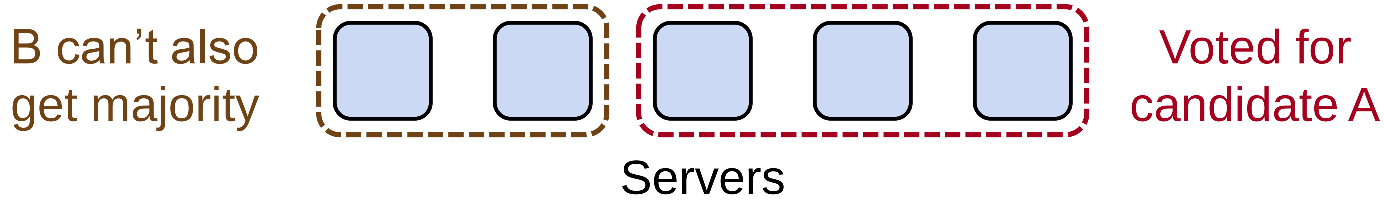
Election Basics

Upon election timeout:

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

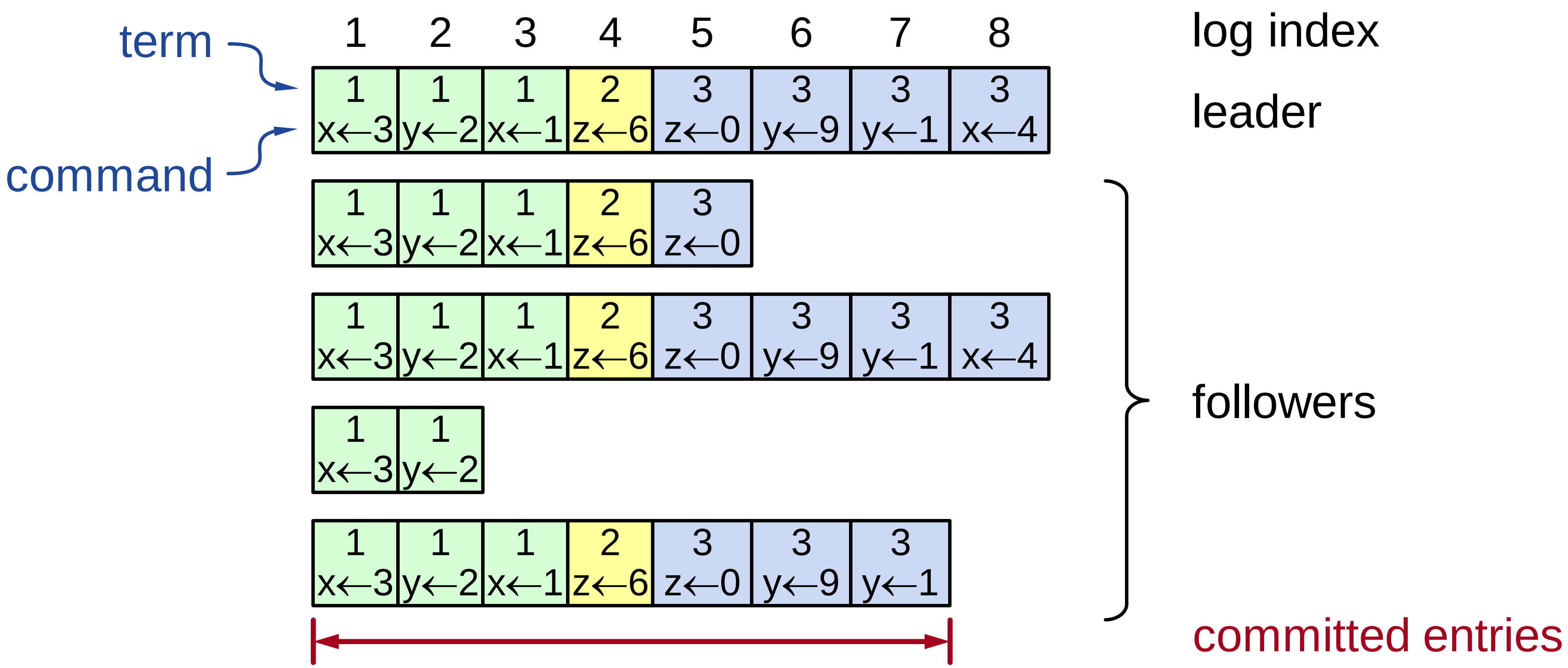
Election Properties

- **Safety: allow at most one winner per term**
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term



- **Liveness: some candidate must eventually win**
  - Choose election timeouts randomly from, e.g., 100-200ms range
  - One server usually times out and wins election before others wake up

Log Structure



- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**

Normal Operation

- **Client sends command to leader**
- **Leader appends command to its log**
- **Leader sends AppendEntries RPCs to followers**
- **Once new entry safely committed:**
  - Leader applies command to its state machine, returns result to client
- **Catch up followers in background:**
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers apply committed commands to their state machines
- **Performance is optimal in common case:**
  - One successful RPC to any majority of servers

Log Consistency

High level of coherency between logs:

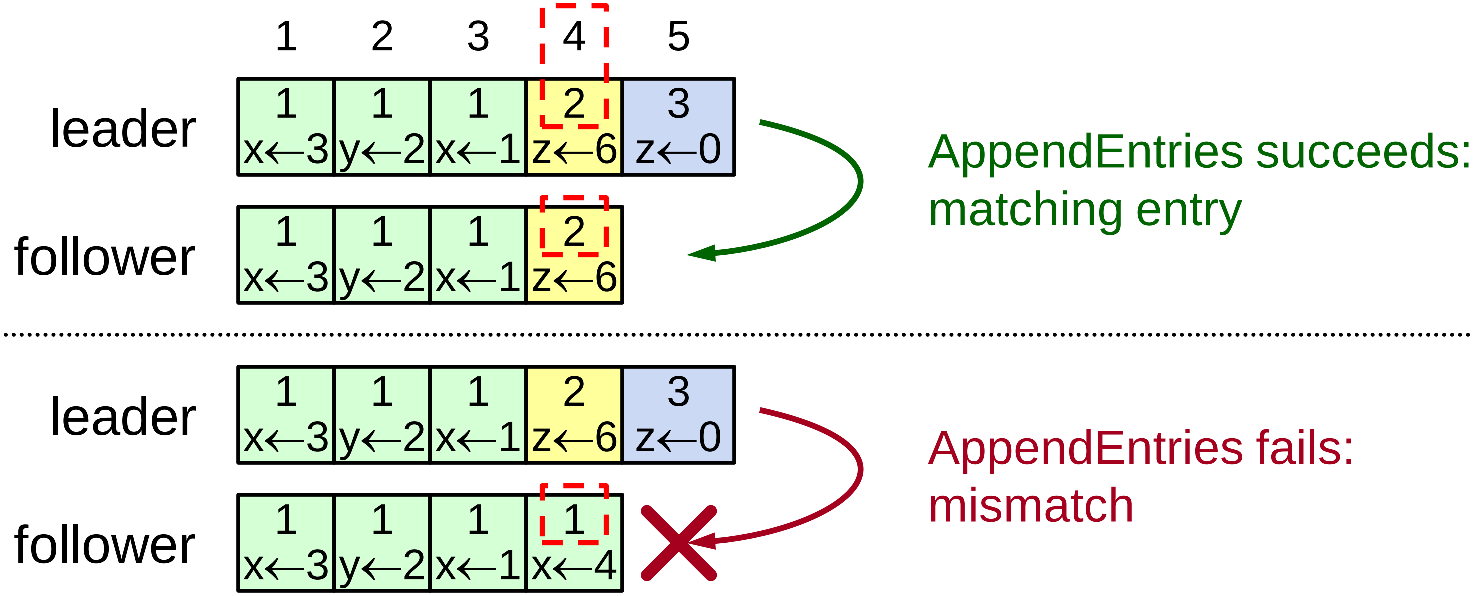
- **If log entries on different servers have same index and term:**
  - They store the same command
  - The logs are identical in all preceding entries

1	2	3	4	5	6
1 x←3	1 y←2	1 x←1	2 z←6	3 z←0	3 y←9
1 x←3	1 y←2	1 x←1	2 z←6	3 z←0	4 x←4

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

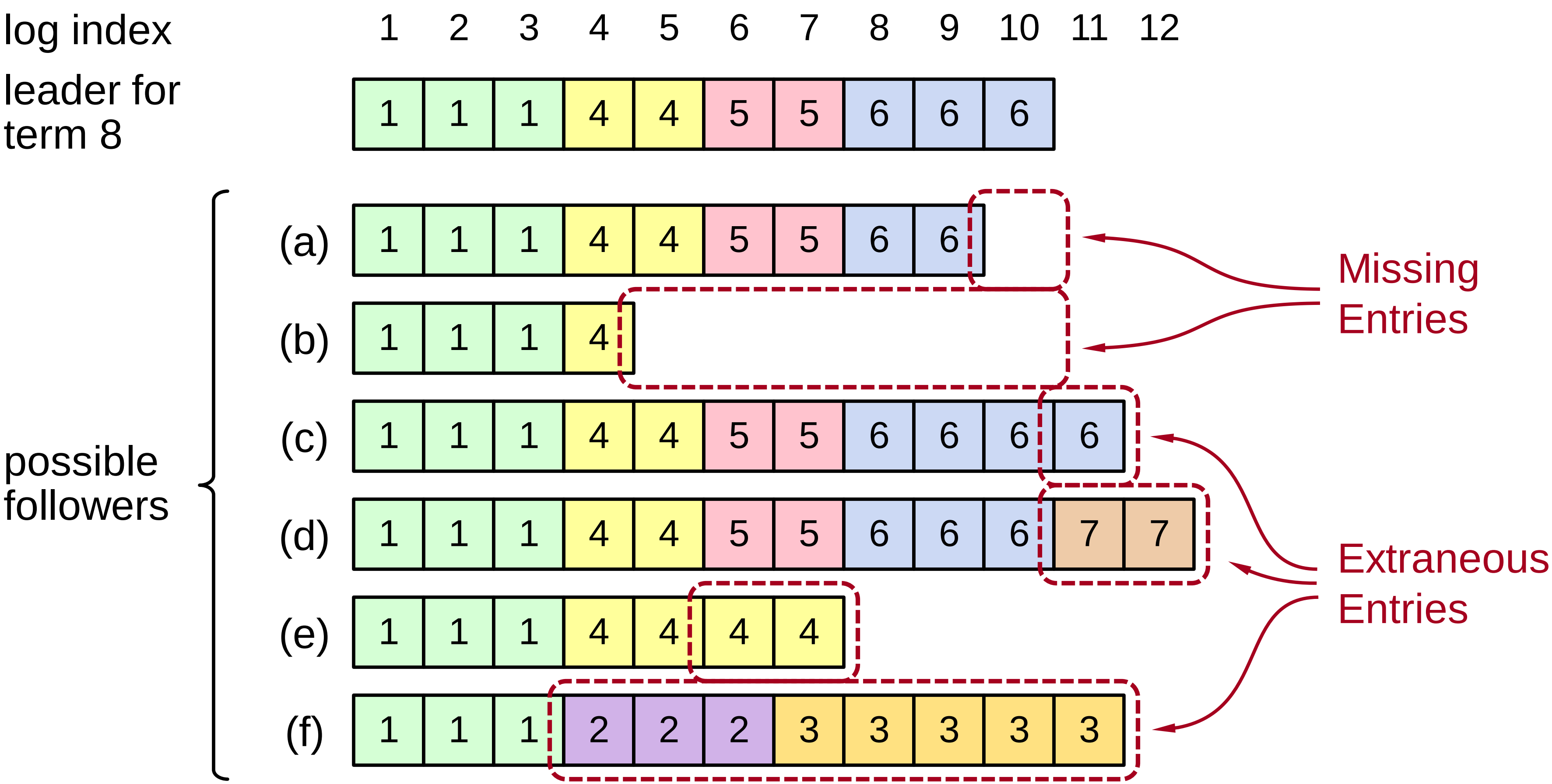
AppendEntries Consistency Check

- **Each AppendEntries RPC contains index, term of entry preceding new ones**
- **Follower must contain matching entry; otherwise it rejects request**
- **Implements an induction step, ensures coherency**



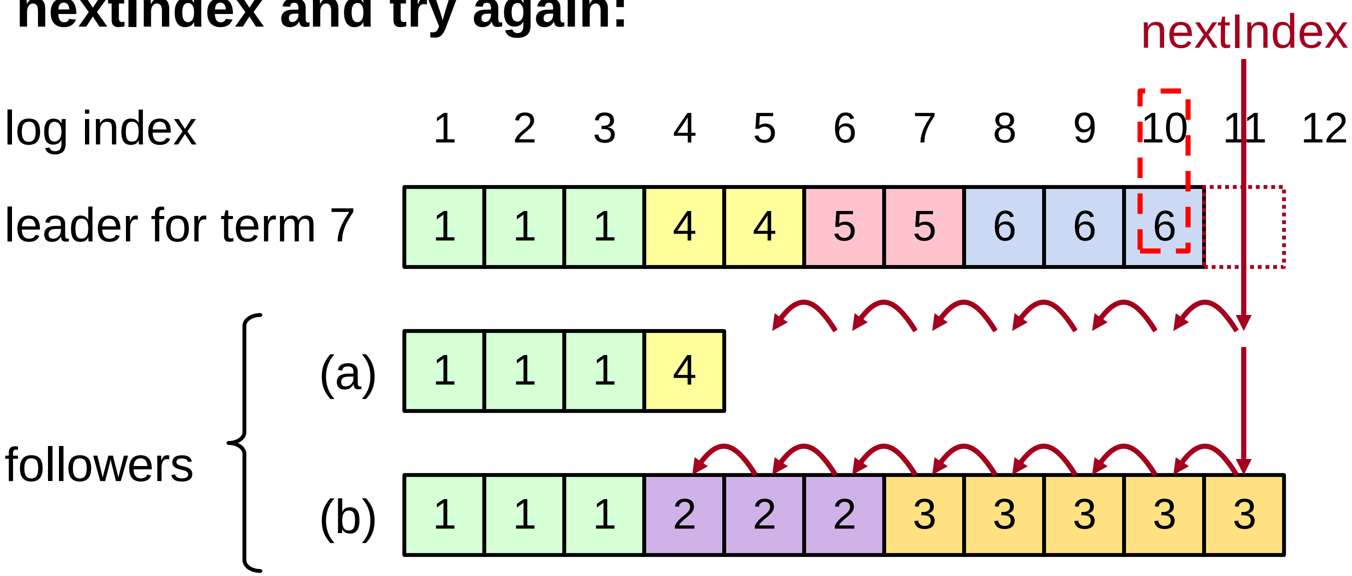
Log Inconsistencies

Leader changes can result in tmp. log inconsistencies:

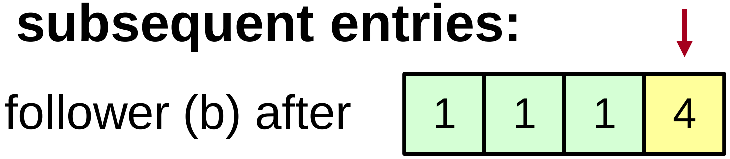


Repairing Follower Logs

- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
- **When AppendEntries consistency check fails, decrement nextIndex and try again:**



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



Safety Requirement

Any two committed entries at the same index must be the same.

Leader marks entry committed ⇨ **Entry present in every future leaders' log**

Restrictions on commitment

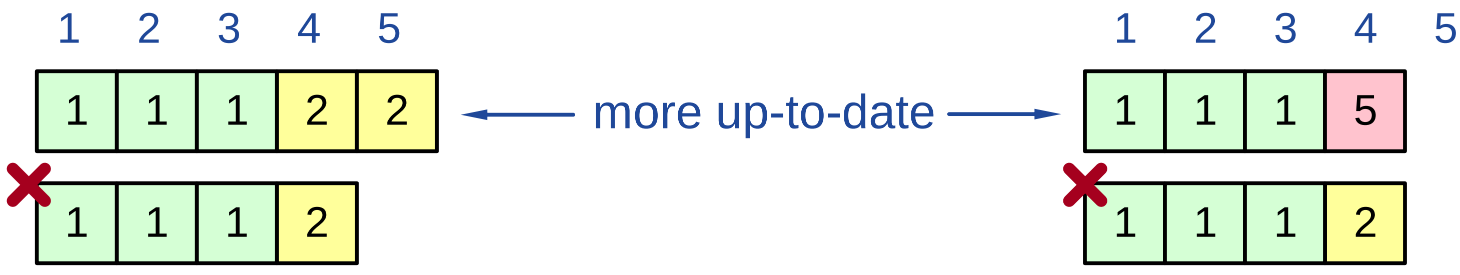
Restrictions on leader election

Picking Up-to-date Leader

- **During elections, candidate must have most up-to-date log among electing majority:**
  - Candidates include log info in RequestVote RPCs (length of log & term of last log entry)
  - Voting server denies vote if its log is more up-to-date:

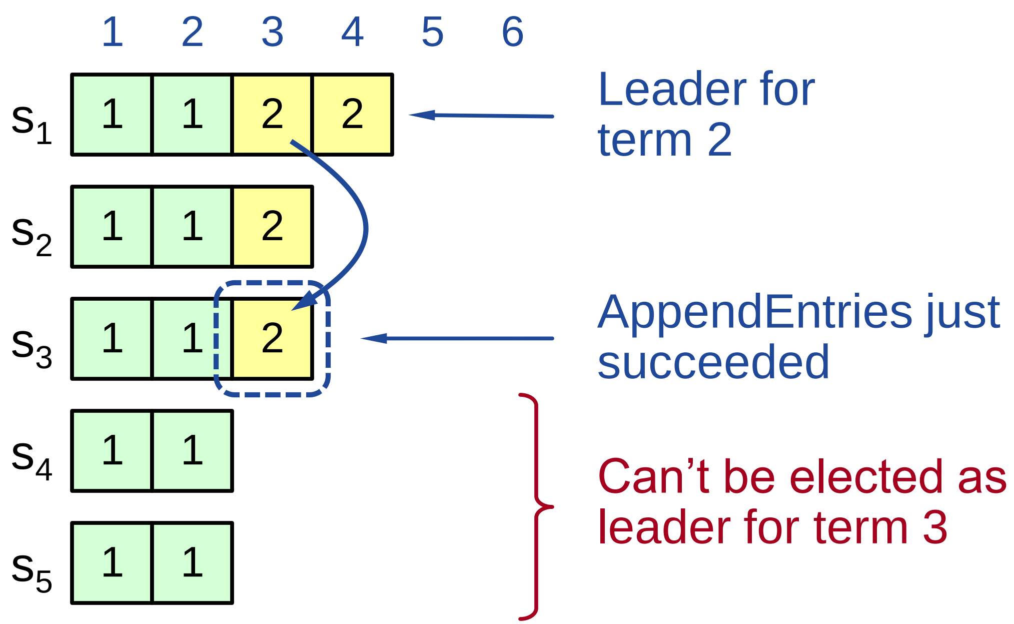
Same last term but different lengths:

Different last terms:



Committing Entry from Current Term

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

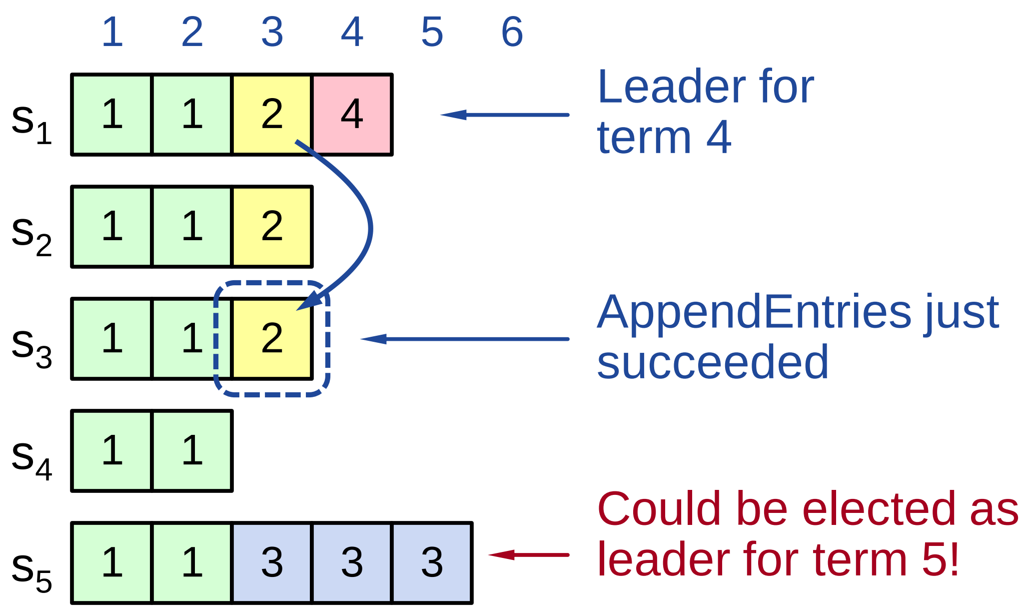


- **Majority replication makes entry 3 safe:**

Leader marks entry committed ⇨ **Entry present in every future leaders' log**

Committing Entry from Earlier Term

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

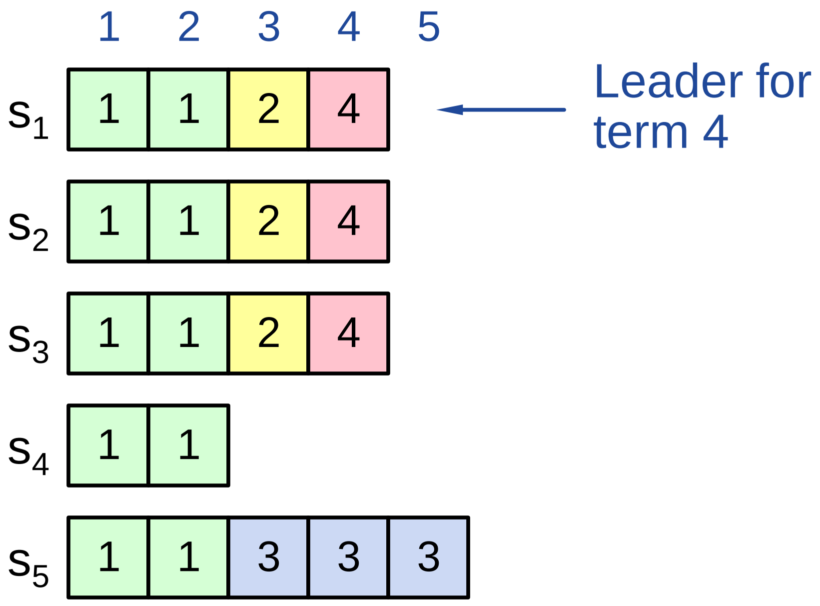


- **Entry 3 not safely committed:**

Leader marks entry committed ⇨ **Entry present in every future leaders' log**

New Commitment Rules

- New leader may not mark old entries committed until it has committed an entry from its current term.
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

Raft Summary

1. **Leader election**
2. **Normal operation**
3. **Safety**

More at http://raftconsensus.github.io:

- Many more details in the paper (membership changes, log compaction)
- Join the raft-dev mailing list
- Check out the 25+ implementations on GitHub

Diego Ongaro @ongardie