Distributed Systems

ECE428

Lecture 14

Adopted from Spring 2021

Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - Block-chains (distributed consensus)

Recap

- Consensus is a fundamental problem in distributed systems.
- Consensus in synchronous systems is possible.
    - Algorithm based on time-synchronized rounds.
    - Need at least (F+1) rounds to handle up to F failures.
- Consensus in asynchronous systems not possible.
    - Cannot distinguish between a timeout and a very slow process.
    - Paxos algorithm:
        - Guarantees safety but not liveness.
        - Hopes to terminate if under good enough conditions.

Paxos Algorithm

- Three types of roles:
  - Proposers: propose values to *acceptors*.
    - All or subset of processes.
    - Having a *single proposer* (leader) may allow faster termination.
  - Acceptors: accept proposed values (under certain conditions).
    - All or subset of processes.
  - Learners: learns the value that has been accepted by *majority* of acceptors.
    - All processes.

Paxos Algorithm

- Key condition:
  - When majority of acceptors accept a single proposal with a value v, then that value v becomes the decided value.
    - This is an implicit decision. Learners may not know about it right-away.
  - Any higher-numbered proposal that gets accepted by majority of acceptors after the implicit decision must propose the same decided value.

Paxos Algorithm: Two phases

- Phase 1:
  - A proposer selects proposal number (*n*), sends prepare request with *n* to majority of acceptors requesting:
    - Promise me you will not reply to any other proposal with a lower number.
    - Promise me you will not accept any other proposal with a lower number.
  - If an acceptor receives a prepare request for proposal #*n*, and it has not responded to a prepare request with a higher number, it replies back saying:
    - OK! I will make that promise for any request I receive in future.
    - (If applicable) I have already accepted a value *v* from a proposal with lower number m < n. This proposal has the highest number among the ones I accepted so far.

Paxos Algorithm: Two phases

- Phase 2:
  - If a proposer receives an OK response for its prepare request #n from a *majority* of acceptors, then it sends an accept request with a proposed value. What is the proposed value?
    - The value v of the *highest numbered proposal* among the received responses.
    - Any value if no previously accepted value in the received responses.
  - If an acceptor receives an accept request for proposal #*n,* and it has not responded a prepare request with a higher number, it accepts the proposal.
- *What if proposer does not hear from majority of acceptors?*
  - Wait for some time, then issue a new request with a higher number.

Paxos Algorithm

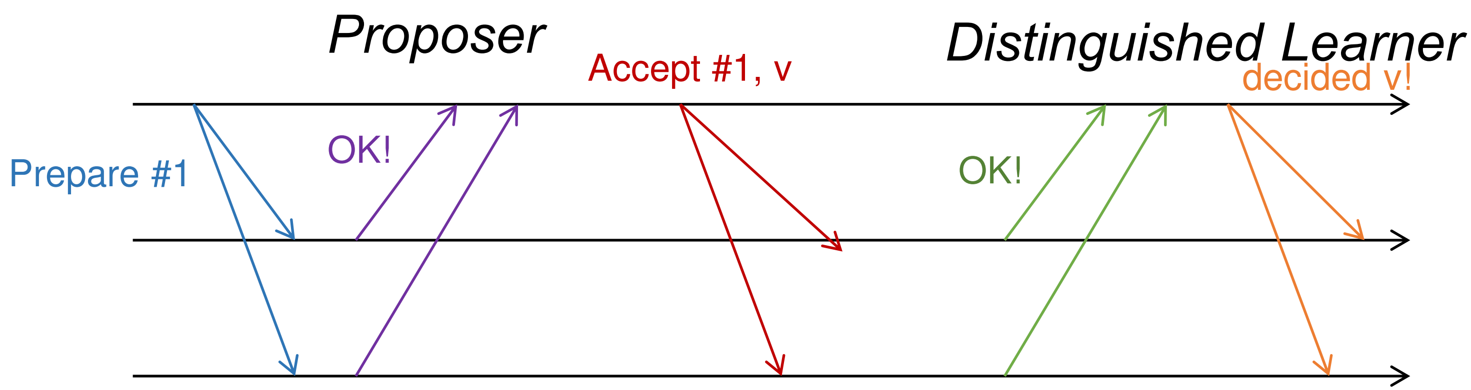
- When majority of acceptors accept a single proposal with a value *v*, then that value v becomes the *decided* value.
  - Suppose this proposal has a number *m*.
  - By design of the algorithm: *any subsequent proposal with a number n higher than m will propose a value v.*
  - Proof by induction:
    - Induction hypothesis: every proposal with number in [m,…..n-1] proposes value *v*.
    - Consider a set C with majority of acceptors that have accepted m's proposal (and value *v*).
    - Every acceptor in C has accepted a proposal with number in [m,…..n-1] .
      - Every acceptor in C has accepted a proposal with value *v*.
    - Any set consisting of a majority of acceptors has at least one member in C.
      - Proposal #n's prepare request will receive an OK reply with value v.

Paxos Algorithm

- When majority of acceptors accept a single proposal with a value *v*, then that value v becomes the *decided* value.
- How do learners learn about it?
  - Every time an acceptor accepts a value, send the value and proposal # to a *distinguished learner*.
  - This *distinguished learner* will check if a decision has been reached and will inform other learners.
    - When it receives the same value and proposal # from a majority of acceptors.
  - Use a set of distinguished learners to better handle failures.
  - What happens if a message is lost or all distinguished learners fail?
    - May not know that a decision has been reached.
    - A proposer will issue a new request (and will propose the same value). Acceptors will accept the same value and will notify the learner again.

Paxos Algorithm

- Best strategy: elect a single leader who proposes values.
- Assume this leader is also the distinguished learner.



- What if we have multiple proposers? (leader election is not perfect is asynchronous systems)
  - May have a livelock! Two proposers keep pre-empting each-other's requests by constantly sending new proposals with higher numbers.
  - Safety is still guaranteed!

Paxos Algorithm

- What if majority of acceptors fail before a value is decided?
  - Algorithm does not terminate.
  - Safety is still guaranteed!

- What if a process fails and recover again?
  - If it is an acceptor, it must remember highest number proposal it has accepted.
    - Acceptors log accepted proposal on the disk.
  - As long as this state can be retrieved after failure and recovery, algorithm works fine and safety is still guaranteed.

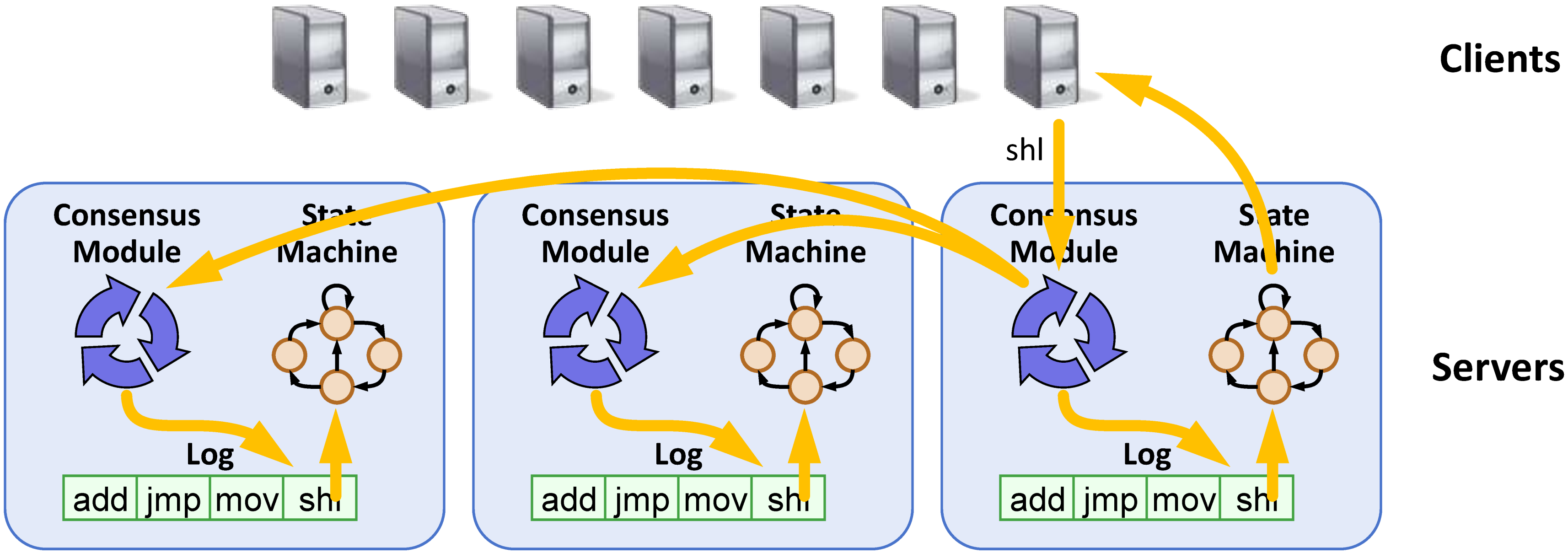
- *Exercise: think about what else can go wrong and how would Paxos handle that situation?*

Log Consensus

- Paxos algorithm (discussed so far) is used for deciding on a single value.

- Many practical systems need to decide on a sequence of values (log).

Replicated Log



- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

Log Consensus

- Paxos algorithm (discussed so far) is used for deciding on a single value.
- Many practical systems need to decide on a sequence of values (log).
- Multi-Paxos: run Paxos repeatedly for each log entry.
  - Quickly becomes very complex.
  - Performance optimizations further increase the complexity.

Paxos is 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 is 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

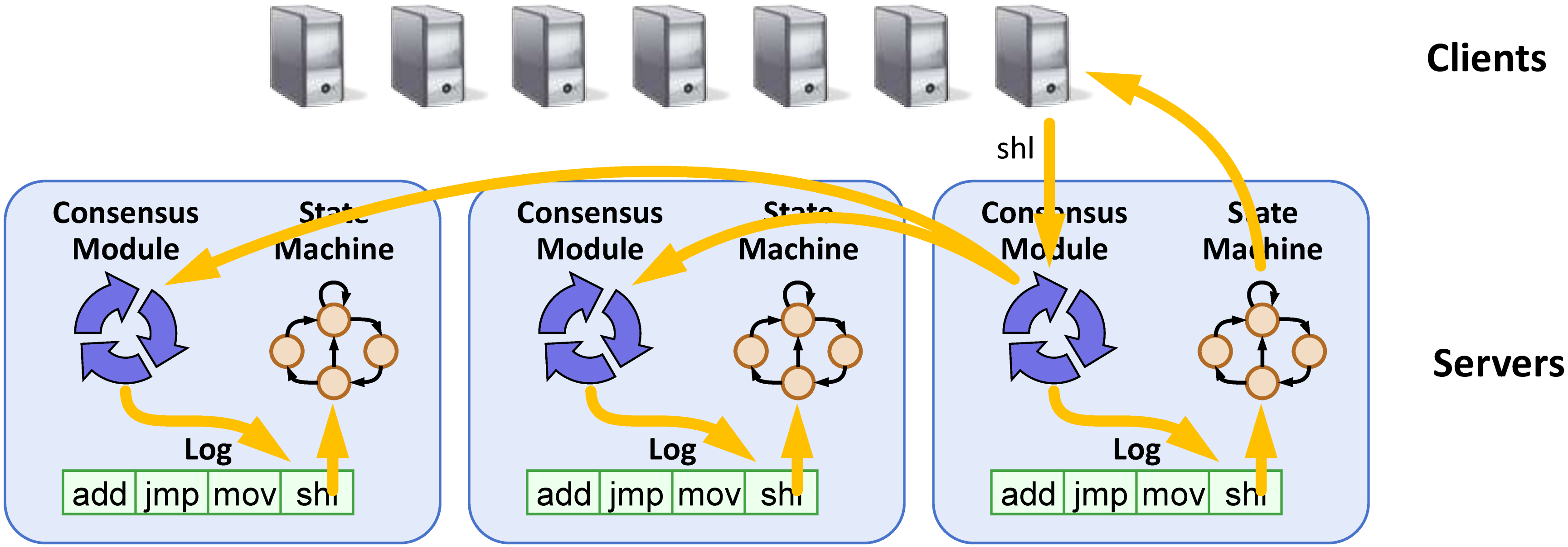
Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - Block-chains (distributed consensus)

Raft: A Consensus Algorithm for Replicated Logs

Slides from Diego Ongaro and John Ousterhout,
Stanford University

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 majority of servers are up
- Failures: fail-stop (not Byzantine), delayed/lost messages

Goal: Design for understandability

- Main objective of Raft's design
  - 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.

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 (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders

Raft Overview

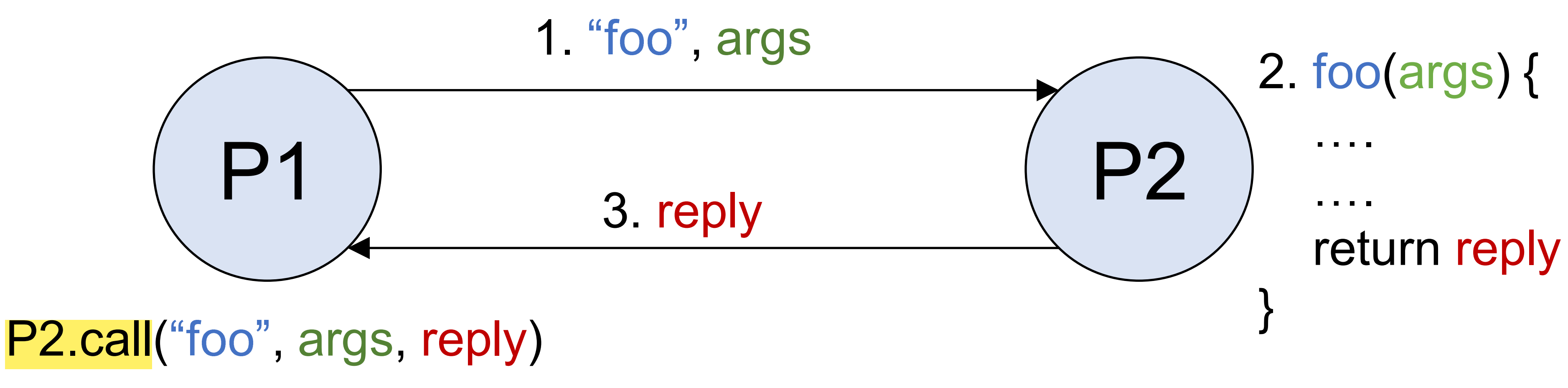
1. Leader election:
    - Select one of the servers to act as leader
    - Detect crashes, choose new leader
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders

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: issues no RPCs (requests), responds to incoming RPCs
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

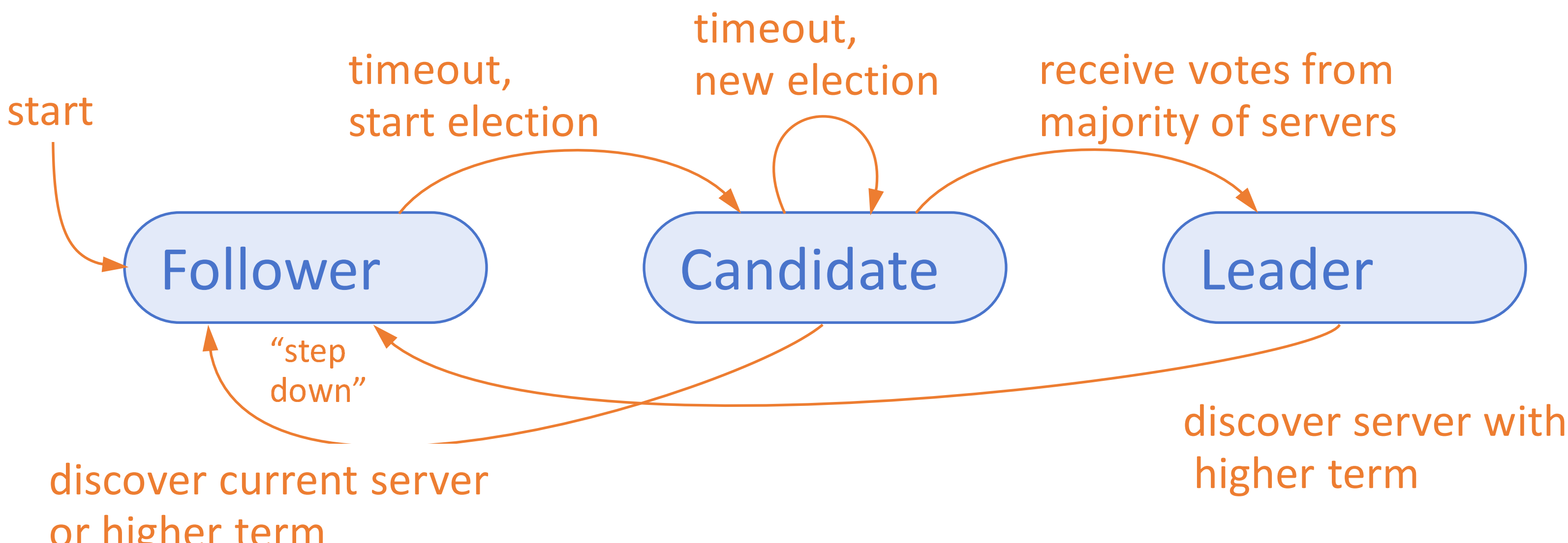
Quick Detour: RPCs

- Raft servers communicate via RPCs.
- What are RPCs?
  - Remote Procedure Calls: *procedure call between functions on different processes*
  - *Convenient programming abstraction.*

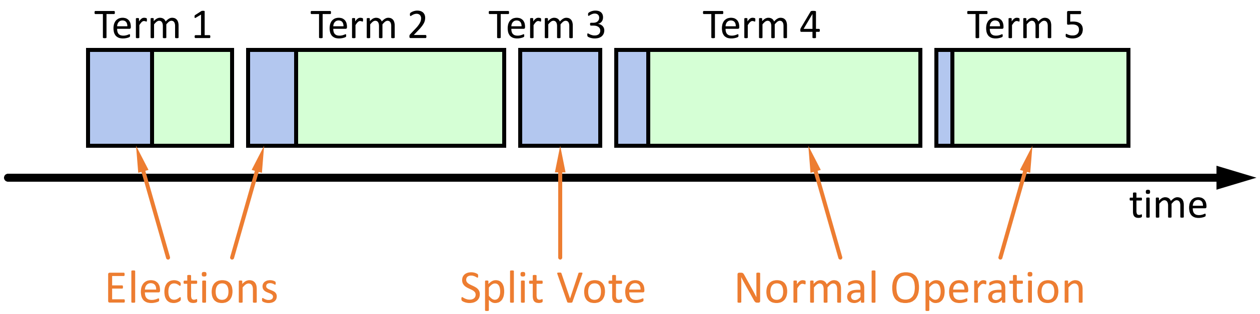


Server States

- 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: issues no RPCs, responds to incoming RPCs
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers



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

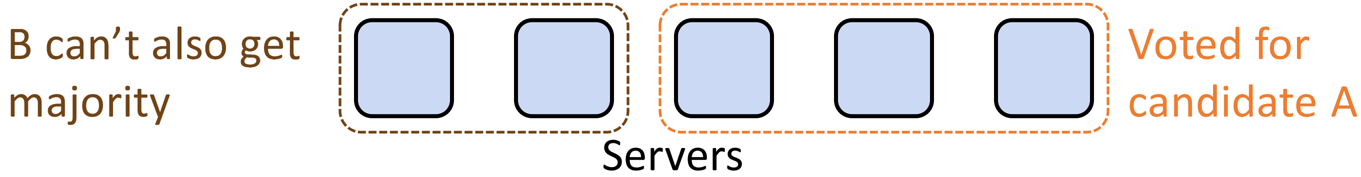
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts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

Election Basics

- On timeout:
  - Increment current term
  - Change to Candidate state
  - Vote for self
  - Send RequestVote RPCs to all other servers:
    1. Receive votes from majority of servers:
       - Become leader
       - Send AppendEntries heartbeats (RPCs) 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

Elections, cont'd

- 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*

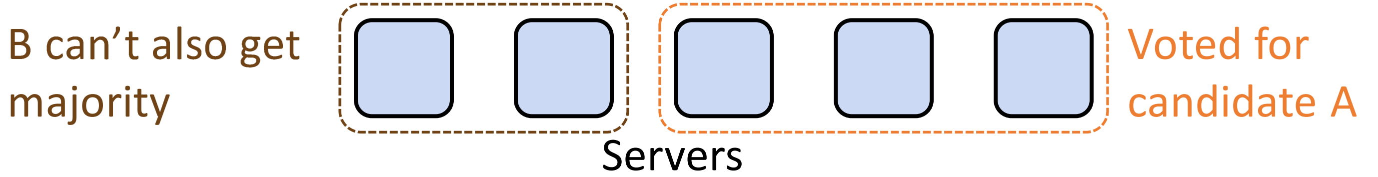


- Liveness: some candidate must eventually win

- *Safety is guaranteed. Liveness is not.*
  - *Election may result in a split vote – no candidate gets majority.*

Elections, cont'd

- 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 in [T, 2T]
  - One server usually times out and wins election before others wake up. Works well if T >> broadcast time
- *Safety is guaranteed. Liveness is not.*
  - *Election may result in a split vote – no candidate gets majority.*

Next Class

- Visualizations to better leader election with Raft.
- Raft’s log replication algorithm.