PROJECT GOAL: Mini Raft

Implement a highly available and fault tolerant SMR based on Raft protocol.

As long as the majority of nodes are alive:

- The client can issue read and write requests to live nodes.
- SMR safety: The committed log should not be reversed or lost.

Assumptions

- We do not assume byzantine failure.

- Types of failure considered:
  - Node crash
  - Network delay

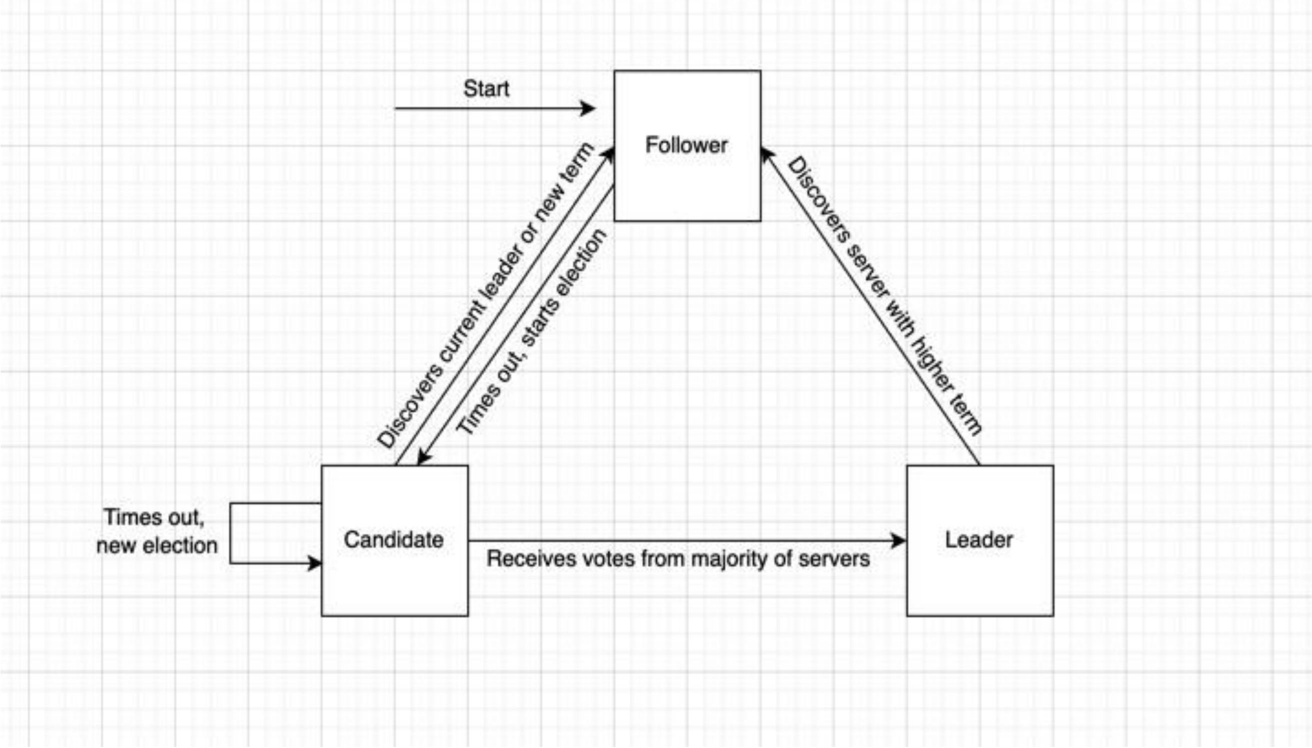
- We use TCP connection.

Two Interconnected Modules

- **Leader Election**

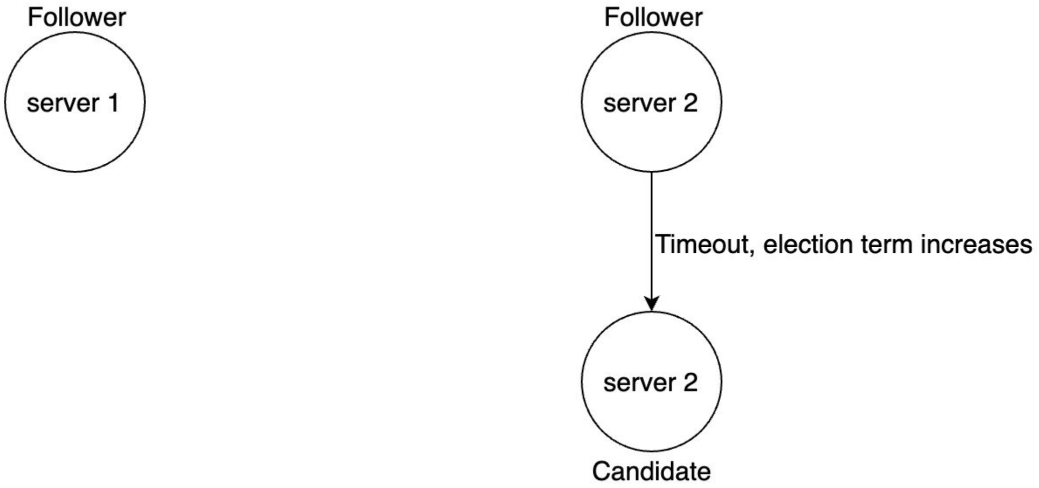
- **Log Replication**

Raft Protocol: Leader Election



Raft Protocol: Leader Election

To avoid the probability of having multiple followers becoming candidate at the same time, randomize the election timeout duration for each server.



Raft Protocol: Leader Election

LEADER ELECTION

- RequestVote RPC:

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:

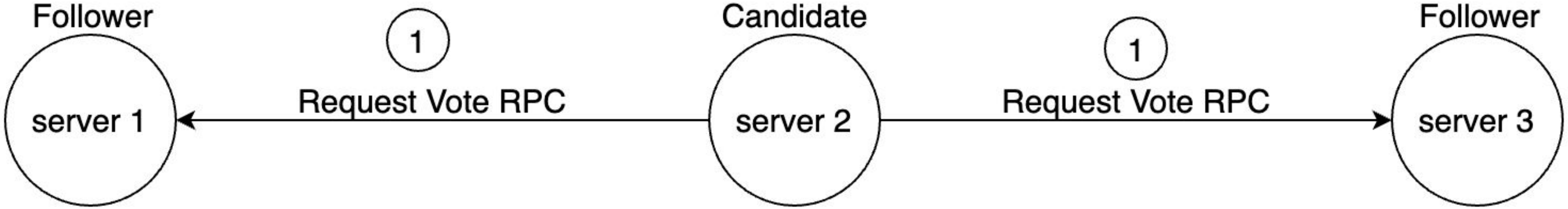
term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

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

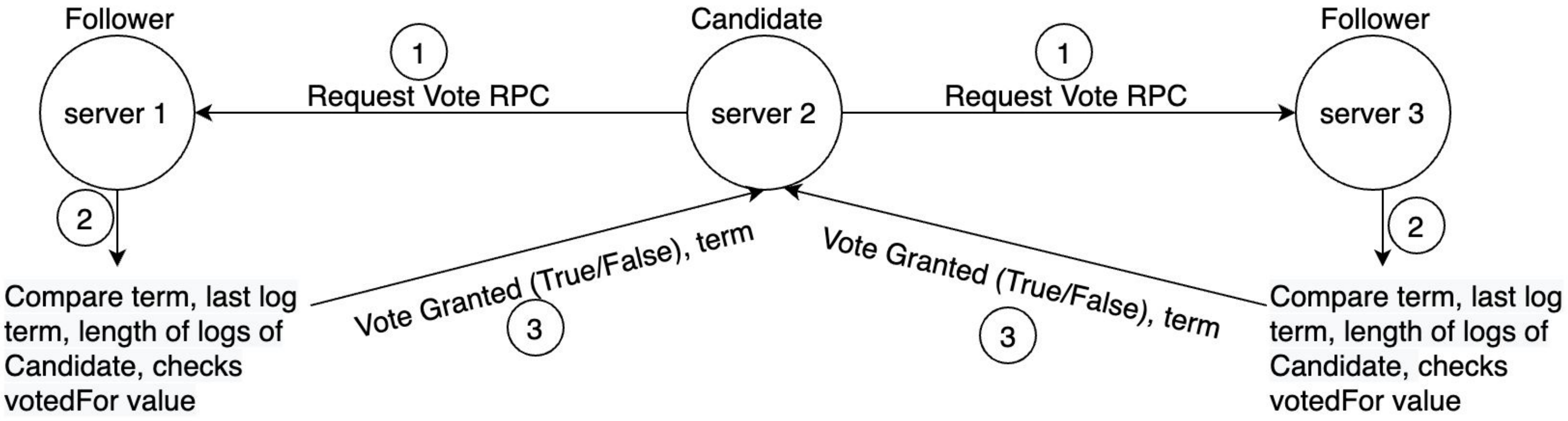
Raft Protocol: Leader Election

Candidate Perspective:



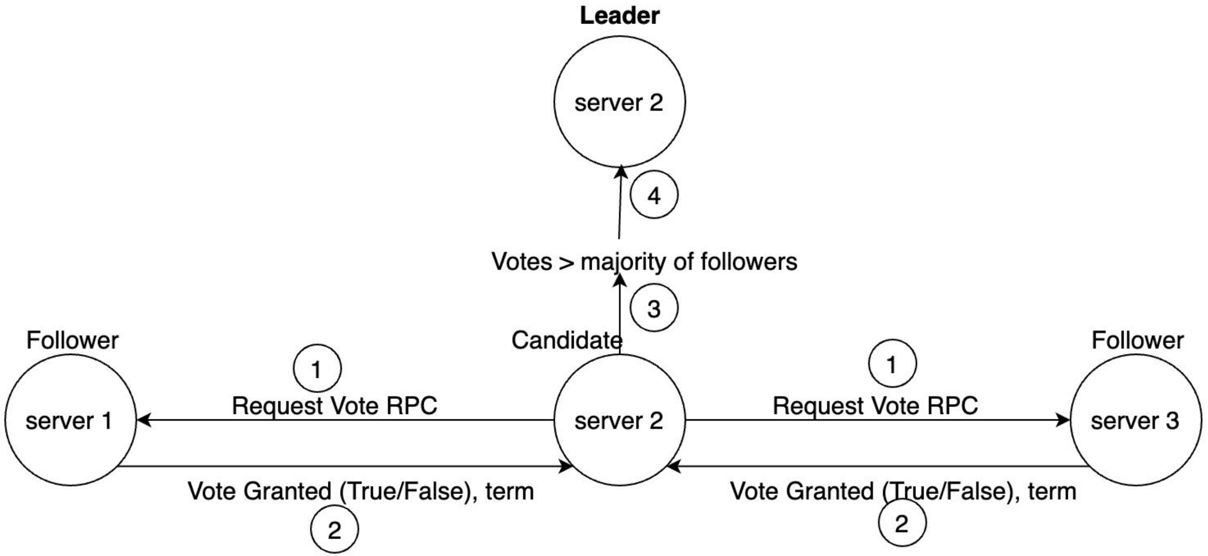
Raft Protocol: Leader Election

Follower perspective:



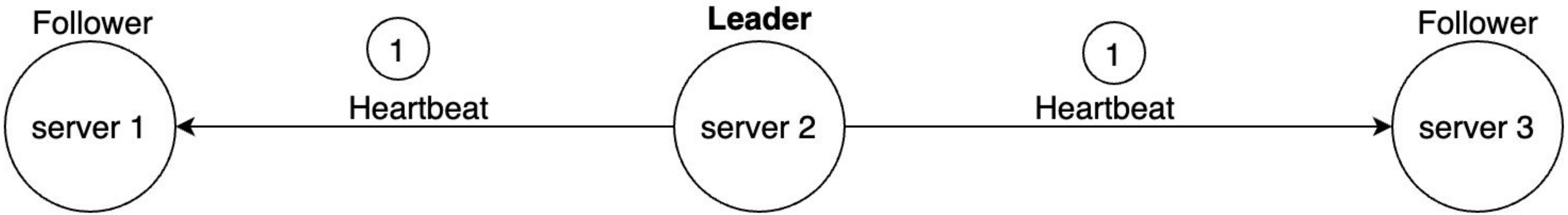
Raft Protocol: Leader Election

Candidate Perspective:



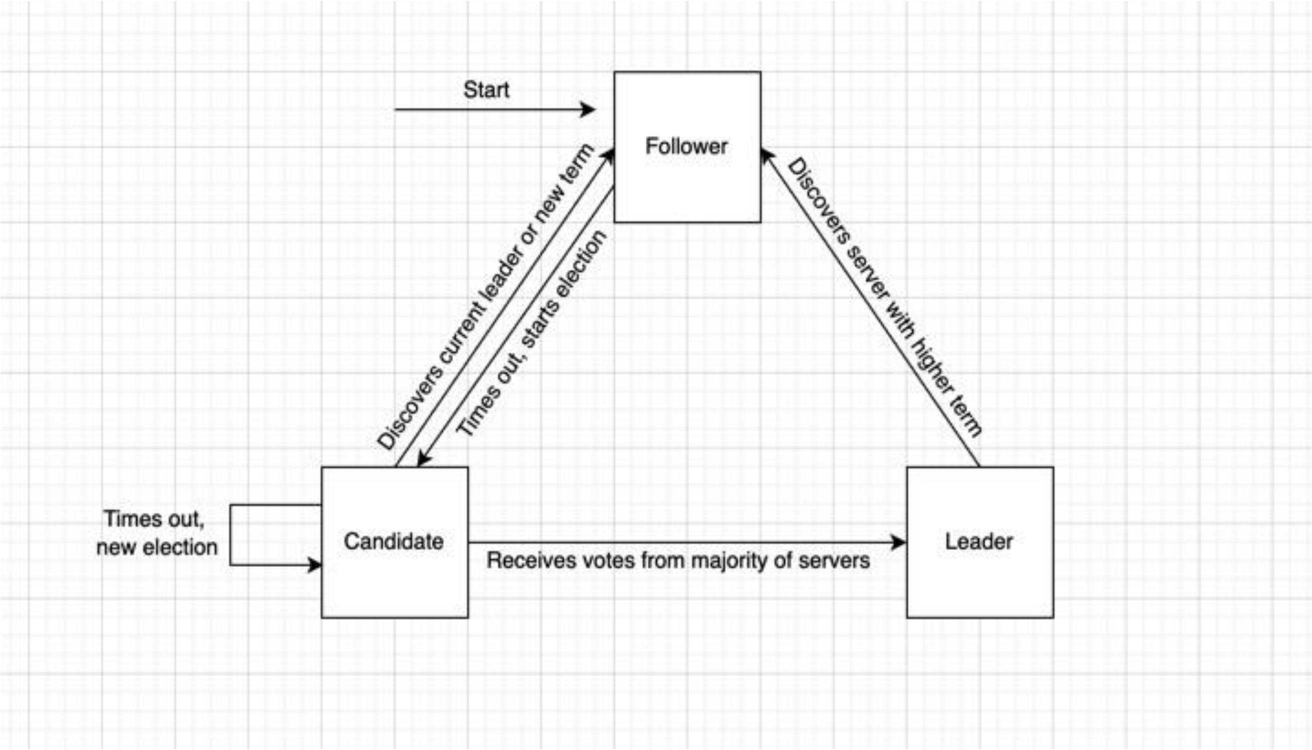
Raft Protocol: Leader Election

Leader perspective:

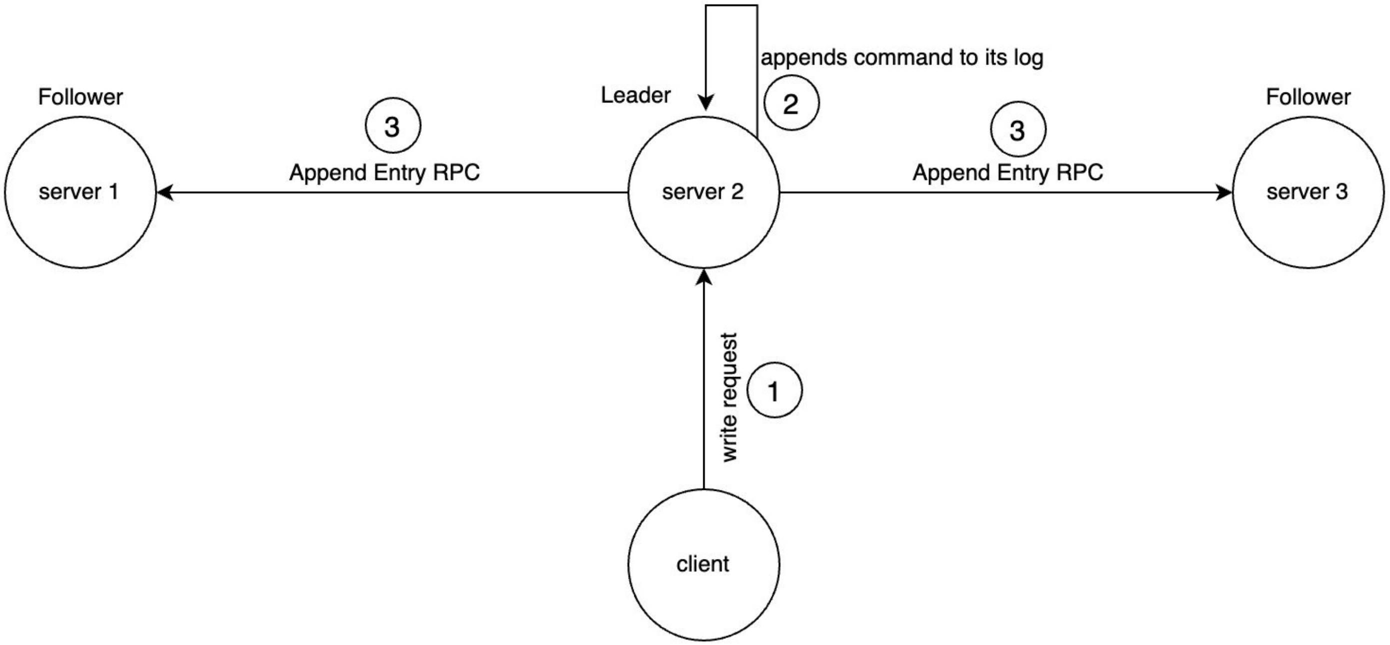


Raft Protocol: Leader Election

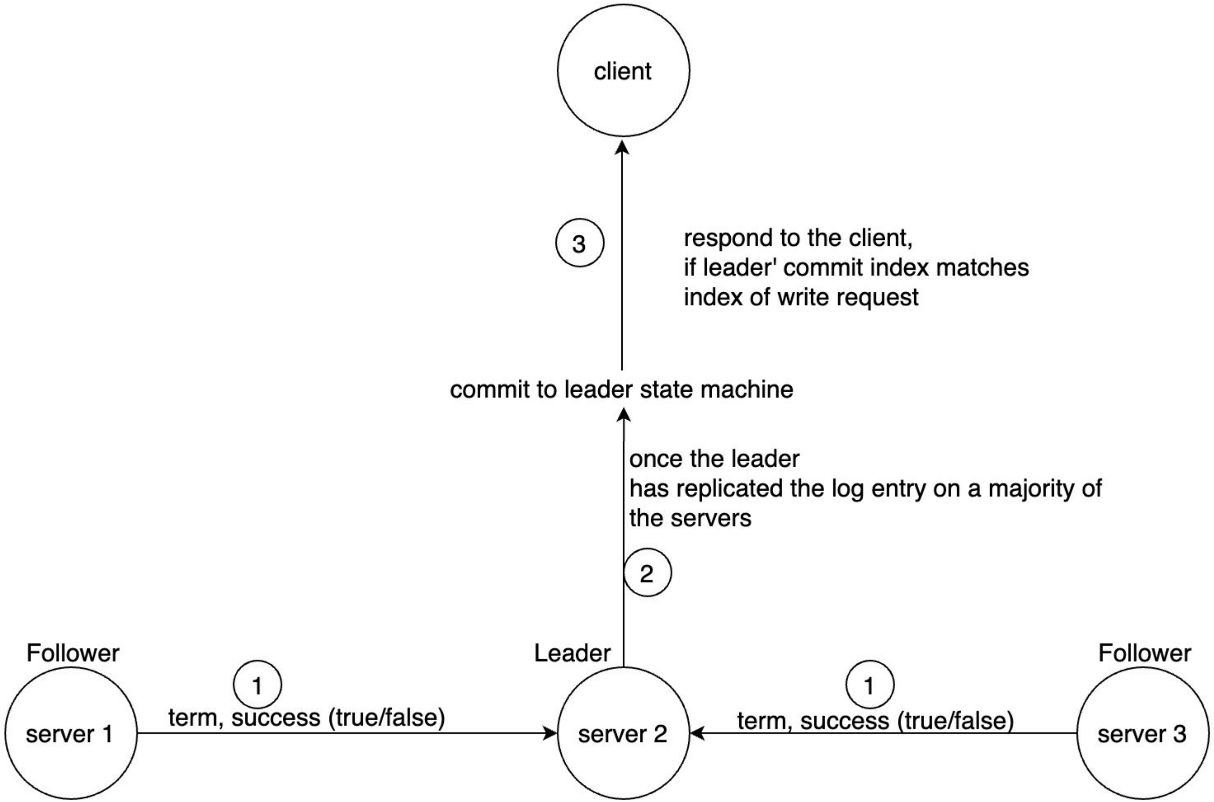
Recap:



Raft Protocol: Log Replication Control Flow



Raft Protocol: Log Replication Control Flow



Software Design: Log Replication

AppendEntries RPC

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

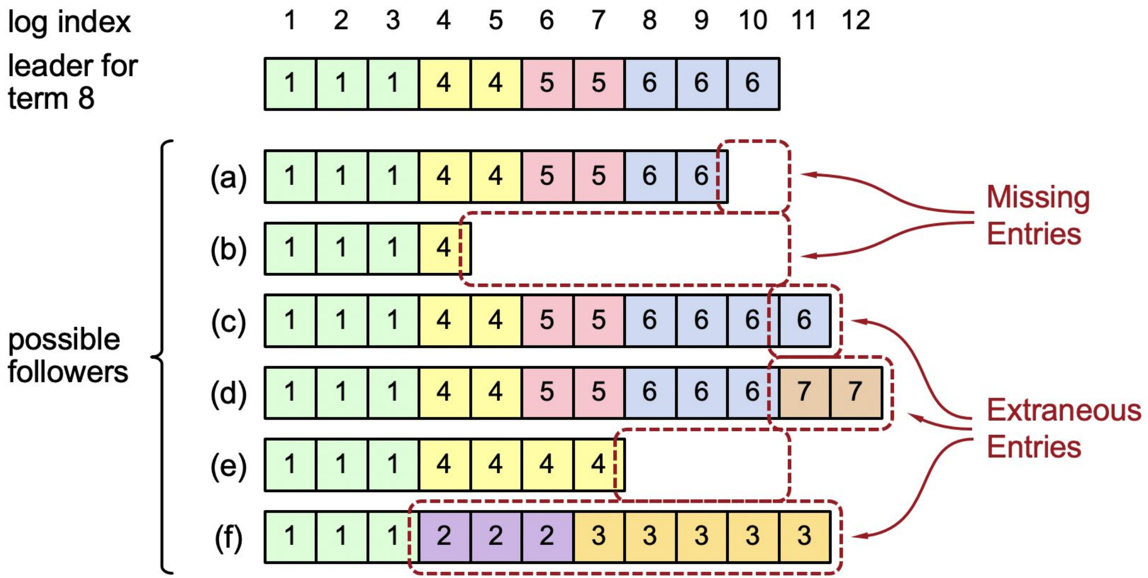
Arguments:

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

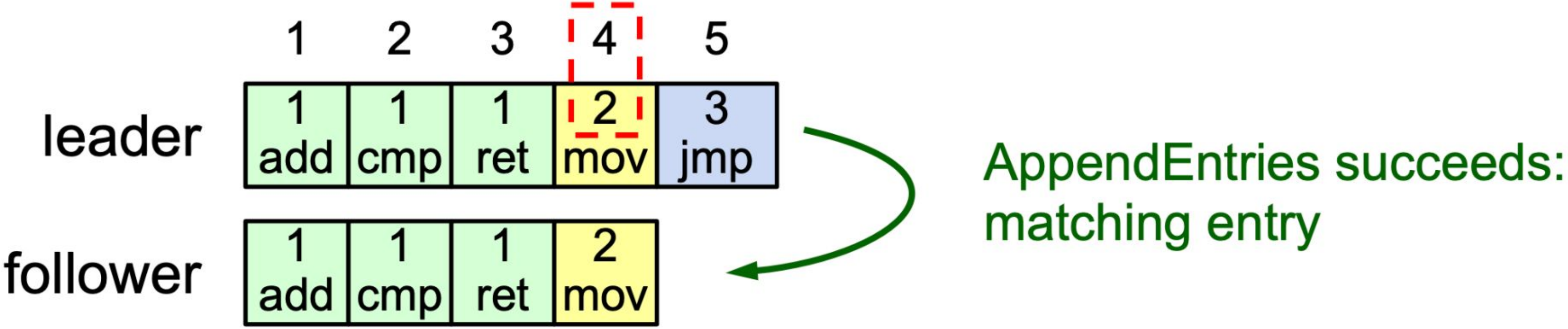
Raft Protocol: Log Replication Complexity



Raft Protocol: Log Replication

The follower approves append entry request:

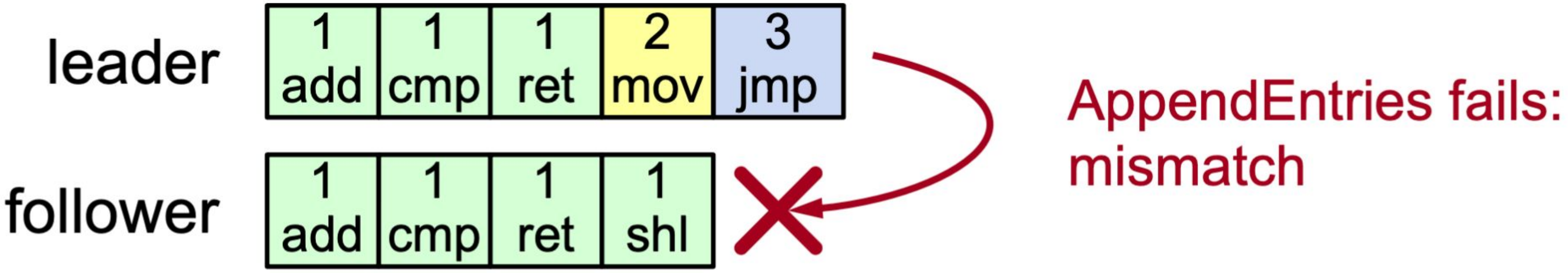
- If its log contains an entry at prevLogIndex whose term matches prevLogTerm



LOG REPLICATION

The follower reply false:

- If its log does not contain an entry at prevLogIndex whose term matches prevLogTerm



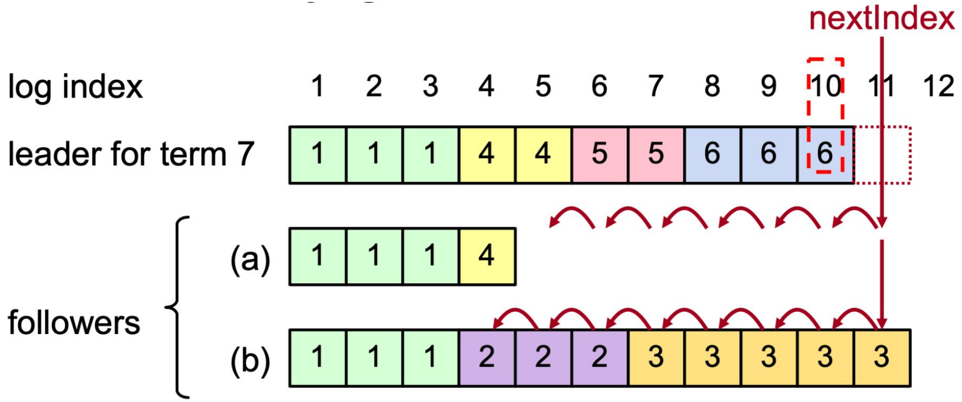
Raft Protocol: Log Replication

Raft Protocol guarantees that the following invariant:

- If log entries on different servers have same index and term, then they store the same command, and the logs are identical in all preceding entries

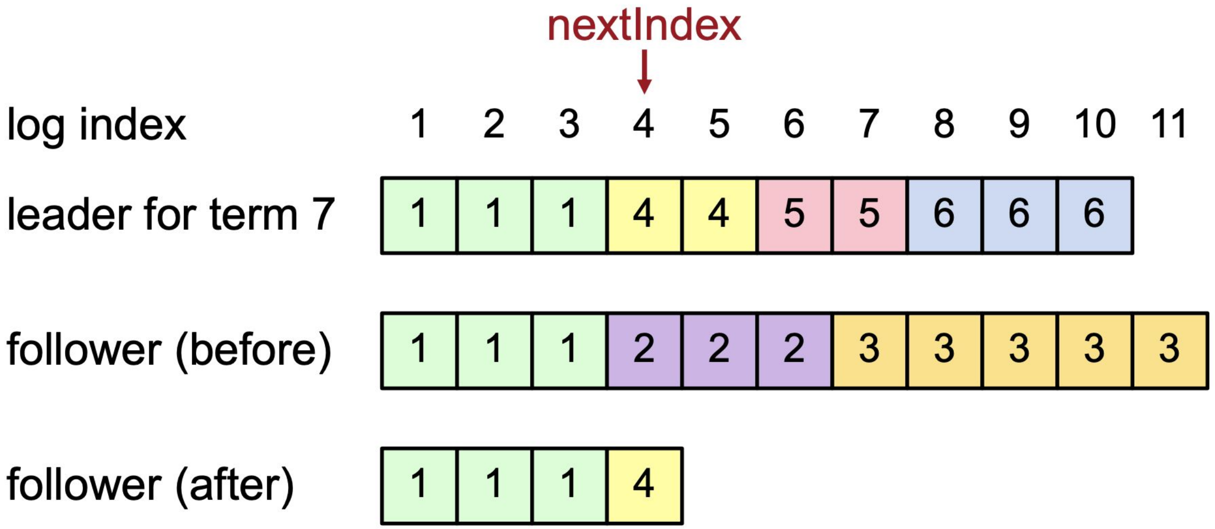
Raft Protocol: Leader Help Straggler Catches Up

- Leader maintains **nextIndex** for each server
- **nextIndex** = index of next log entry to send to that follower
  - (Initialized to leaders last log index + 1)
- When append entry consistency check fails, leader decreases **nextIndex** for that follower and tries again



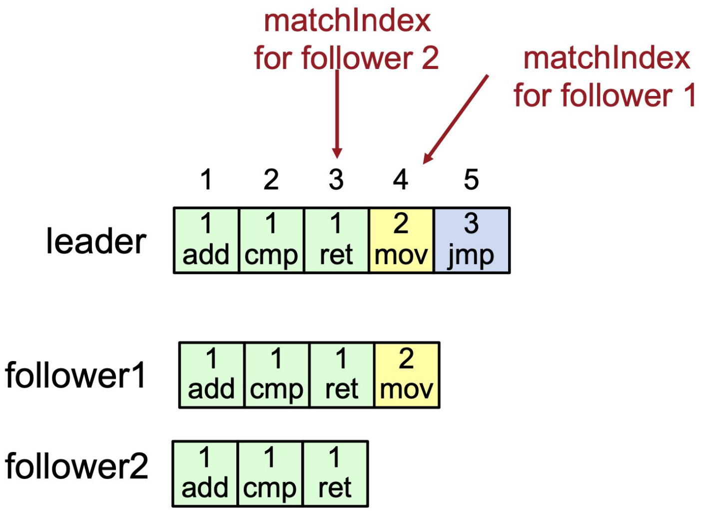
Raft Protocol: Leader Help Straggler Catches Up

Follower overwrites inconsistent entry, it deletes all subsequent entries:



How Raft decide when an entry is committed

- **MatchIndex:**
  - for each server, index of highest log entry known to be replicated on server



How Raft decide when an entry is committed

- **Updating the commit index of the leader:**

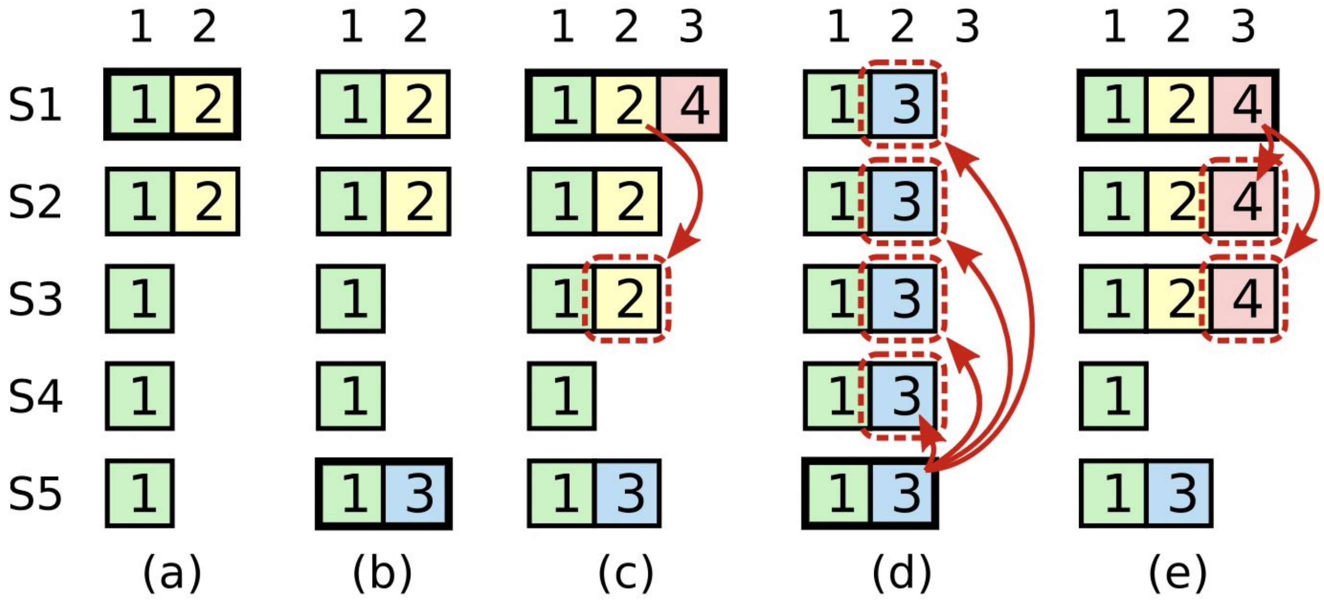
  If there exists an N such that N > commitIndex such that:

  1. a majority of matchIndex[i] ≥ N, and
  2. log[N].term == currentTerm:

  Update commitIndex = N

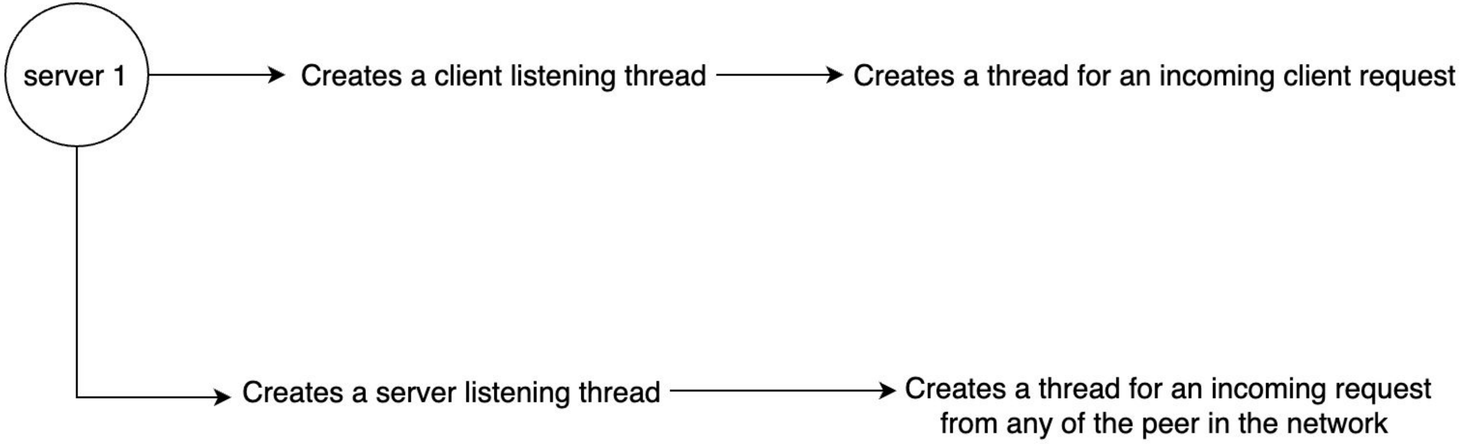
Raft Protocol: When a New Leader is Elected

Raft never commits log entries from previous terms by counting replicas.



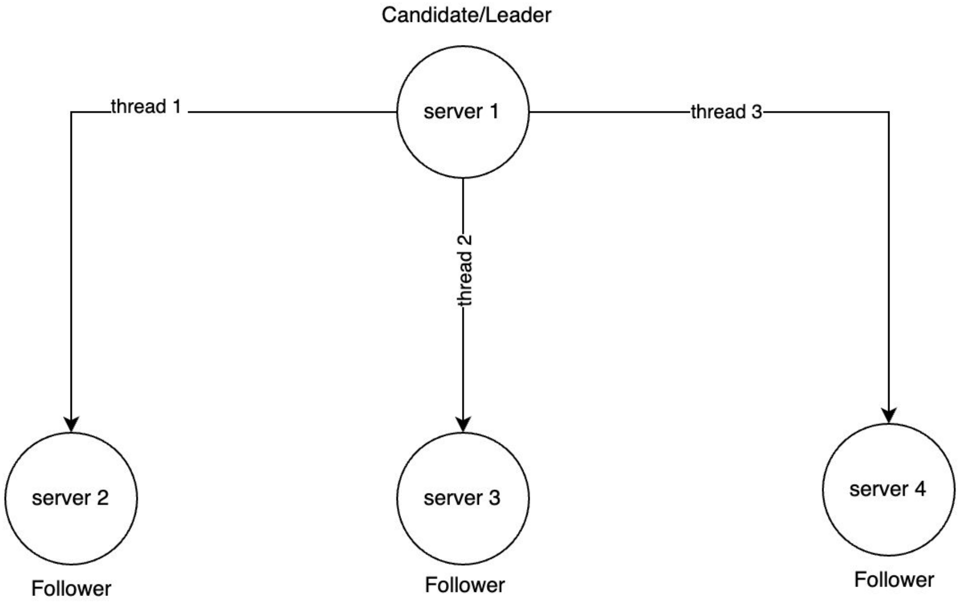
Our Implementation: Raft Server

- Multithreading
- Each server maintain one listening port for connection from peer server and for connection from client



Our Implementation: Raft Server

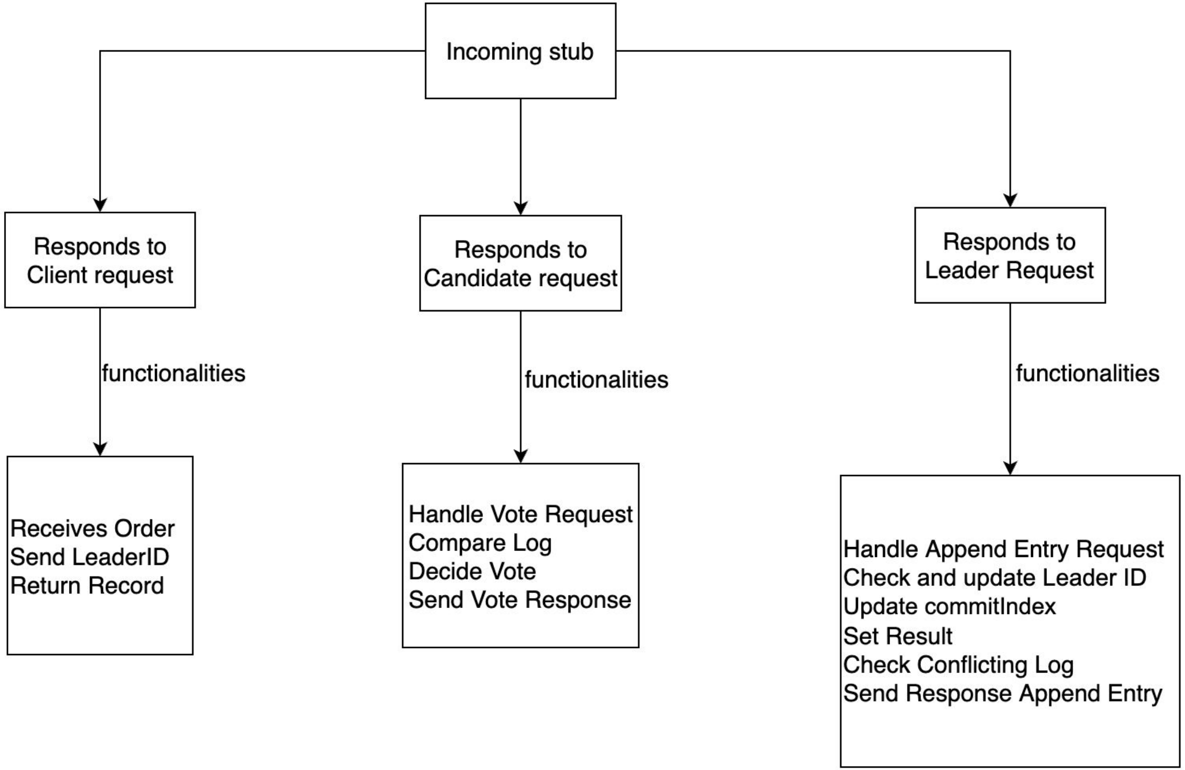
Whenever a server becomes a candidate or leader, it spins a thread to connect to each peer in the network



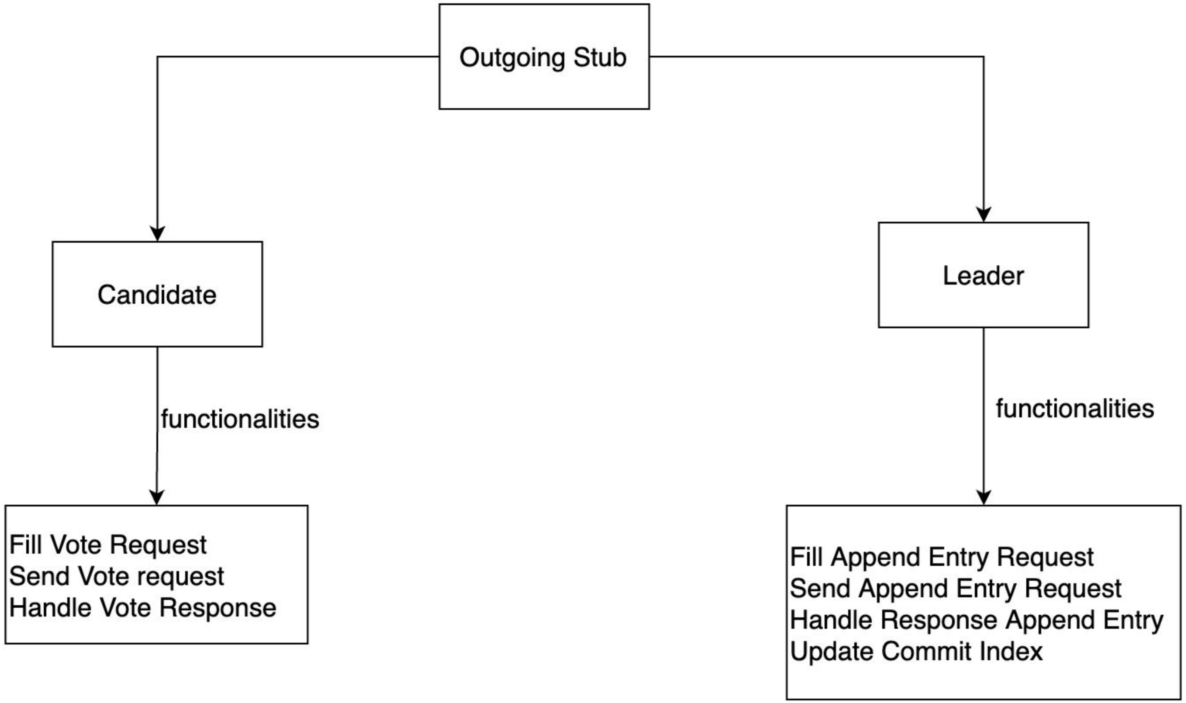
Our Implementation: Raft Server

- InStub: handle incoming connection
  - Used by servers of any role
- OutStub: handle outgoing connection
  - Used by candidate and leader

Our Implementation: InStubs



Our Implementation: OutStub



Evaluation: Leader Election Correctness

Goal:

- Only 1 leader is elected within one term
- Once the server is elected to be a leader, it will continue to be the leader if we do not kill it.
- If we kill the leader, a new leader is elected in the next term immediately.

Testing Helper Tools:

- To really test edge cases, we can initialize the server to take any role through command line argument.

- The client can issue request to server to ask for what it thinks the leader ID currently is.

Test Case 1: Leader Election Fault Tolerant

1. Start 5 server nodes as a follower.

2. Once a leader got elected. Use client program to ask for Leader ID to each individual server to make sure everyone agrees only on leader. Kill the leader.

3. Repeat step 2 until only 2 nodes are alive.

4. Check that no leader is elected, as you would require a majority of 3 votes to become a leader

5. We bring the other three servers back as followers, and then check that a leader is elected.

Test Case 2: Multiple Candidates Race Condition

- In the setting of 5 servers, initialize 3 candidates and 2 dead servers.
- Only 1 leader is elected. Every node agrees on the same leader ID.

Evaluation: Log Replication Fault Tolerant

- In a 5 server network, once a leader has been elected, use the client to issue write request.
- Meanwhile, use the client to issue read request. Check that the record is correct.
- Kill one follower or the leader, and bring it back as a follower.
- Use the client to issue read request to that node and check that the record has been replicated again.

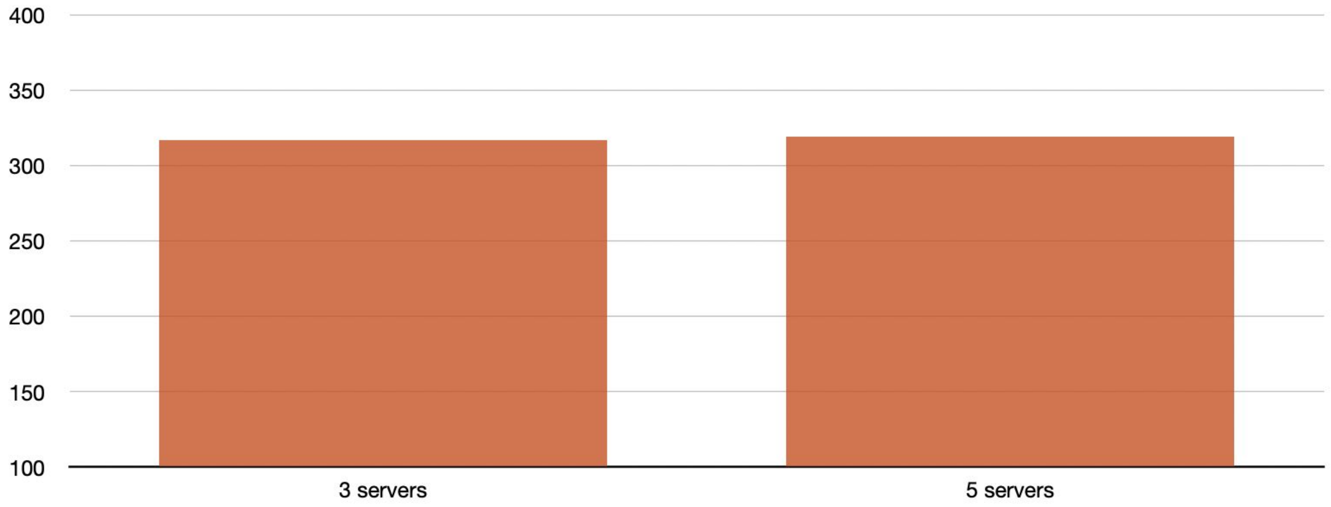
Log Replication Correctness and Safety

Setting: 4 followers and 1 candidate in the beginning of an election

- Initialize the candidate to read logs from an external file that contains only a few log entries, while the follower read logs from a more up-to-date file.
- Started the election, and check that candidate doesn’t become the leader.

Evaluation: Performance

- Mean write latency for two configuration settings: 3 server vs. 5 server with four client threads. Time shown is in ms.



Correctness Under Network Delay

- We simulate network delay by having the node sleep for a randomly amount of time.

- Leader Election and Log Replication modules as usual. Just higher latency, and lower throughput.

Achievement

- Implemented features from assignment 2 such as issuing read and write request in the context of Raft.
    - Nontrivial, because the multithreading design and the fault tolerant protocol is more complex. Deadlock is very common and very hard to debug.

- If at least three servers are alive, the client is guaranteed to find the leader and able to issue write request.

MISSING ITEMS

- Not able to completely implement persisting the server state on a file storage

LEARNINGS

- Multithreading combined with randomized timing makes debugging difficult.
- Learned to divide the whole design into small approachable features and to come up with helper tools to test them.
- We first tried to avoid multithreading by doing asynchronous socket programming using the poll function, but ran into unexpected challenges.
- Resource management is critical in handling fault, especially in multithreading programming. Thankful for the socket class Prof gave in PA1, and **unique_ptr** is very helpful.

QUESTIONS