Raftlike Design Document

Small Implementation of the Raft Consensus System

Author: Bruno Zalcberg
Date: October 2025
Language: Rust

---

1. Architecture Overview

This project implements a simplified Raft consensus algorithm in Rust, featuring automatic leader election, log replication, and crash recovery across a 3-node cluster.

The algorithm is used by a few real-world applications and databases such as Kubernetes (etcd), TiDB

The architecture consists of a few core components:

- HTTP API server (Axum)
- Asynchronous event loop (Tokio)
- Persistent state storage using JSON files
- Command-line interface for interacting with the nodes

1.1. Concurrent Task Architecture

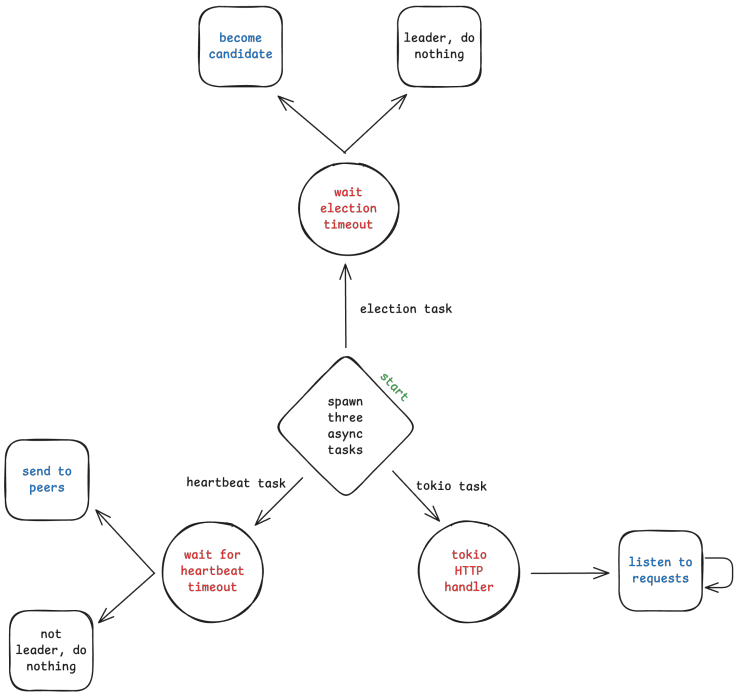

Figure 1: Concurrent task architecture: Election timeout, heartbeat, and HTTP handler tasks operate independently, coordinating through shared state

The system spawns three async tasks at startup that run concurrently. As shown in Figure 1, the three tasks are:

1. **Election Timeout Task** - Followers monitor for leader failures and initiate elections through RPC
2. **Heartbeat Task** - When leader, sends periodic append entries to followers through RPC
3. **HTTP Handler Task** - Accepts and processes client requests

All tasks share access to the `RaftNode` state through `Arc<Mutex<>>`, which ensures that the operations are thread-safe.

1.2. Election Timeout Mechanism

Each follower maintains a randomized election timeout between 300-500ms. This randomization prevents split votes by ensuring nodes start elections at different times.

```
// Pseudo-code representation
timeout = random(300..500ms)
if no_heartbeat_received(timeout):
    become_candidate()
    increment_term()
    request_votes_from_peers()
```

1.3. Voting Rules

A node grants its vote if **all** conditions are satisfied:

1. Candidate's term ≥ node's current term
2. Node hasn't voted for another candidate this term
3. Candidate's log is at least as up-to-date

Log info can also help the system choose new leaders in case of ties or network partitions:

- If last log terms differ → higher term wins
- If terms equal → longer log wins

1.4. Term Management

Terms act as logical clocks and control the system's timeline. Usually, a term is incremented (and an election started) due to network partitions or leader failures.

When a node observes a higher term:

- Immediately steps down to follower (if leader or candidate)
- Updates to new term
- Clears its vote

This prevents stale leaders from disrupting the cluster.

2. Log Replication

Log replication is one of Raft's core mechanisms and is of utmost importance.

2.1. Append Entries Protocol

The leader replicates log entries by sending `AppendEntries` RPCs every 100ms (heartbeat interval).

This request includes the leader's current term, the previous log entry's index and term (for consistency checks), and new log entries.

Consistency check: Follower rejects if `prev_log_index` doesn't match locally.

2.2. Commit Protocol

The leader commits an entry when it is safe to assume that the majority of nodes have replicated it.

```
// Automatic commit detection
for each index in uncommitted_range:
    if majority_has(index) && entry.term == current_term:
        commit_index = index
        apply_to_state_machine()
```

Once committed, the entry is applied to the key-value store.

3. Persistence Strategy

3.1. Persistent State

Three pieces of state survive crashes:

Field	Purpose
`current_term`	Prevents voting in old elections
`voted_for`	Prevents double-voting
`log`	Source of truth for all data

This is enough to recover from failures.

3.2. Storage Format

The node state is serialized to JSON and written to `./states/raft_state_<id>.json` after every modification, such as term changes (elections), vote grants or log appends.

3.3. Recovery Process

On startup (either at start or when recovering from a failure), the node loads its state from the `.json` file and initializes itself as a follower in the last known term.

4. Failure Handling

Detecting failures accordingly is important to ensure the system's reliability and availability.

4.1. Leader Crashes

Followers detect via missed heartbeats (>400ms).

In case a leader is not detected, the follower nodes will start an election when the correspondent timeout expires. That means a new leader will be chosen within 500ms.

The log entries committed on majority survive any leader crashes.

4.2. Split Votes

Two candidates might start elections at the same time. In that case, both might not receive a majority of votes. The election timeout will fire again, but with a different random delay, and the system will eventually converge (within 2-3 rounds)

5. Limitations and Future Improvements

The Raft consensus algorithm is not perfect and has natural limitations. In the context of the CAP theory, it is a CP system (consistent and partition-tolerant).

Consistency (✓): All committed log entries are replicated to a majority before being considered committed. No node can have conflicting committed data.

Partition tolerance (✓): During network partitions, the majority partition elects a new leader and continues operating. The minority partition safely halts write operations.

Availability (×): The system becomes unavailable in two scenarios:
1. During leader elections (~300-500ms window)
2. For minority partitions (cannot achieve quorum)

The algorithm also has a problem with throughput. Raft runs on a single-leader model, which means it centralizes all operations (writes, reads and replication), creating a performance bottleneck.

Not only that, but scaling such a system is also hard, given that the leader workload grows with the total number of nodes.

5.1. Future Improvements

1. Snapshot/compaction for log growth
2. Batched log replication for throughput
3. Metrics dashboard (Prometheus/Grafana)