# 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.

**Core Components:**

- HTTP API server (Axum framework)
- Asynchronous event loop (Tokio runtime)
- Persistent state storage (JSON files)
- Command-line interface

### 1.1. Concurrent Task Architecture

The system spawns three async tasks at startup that run concurrently:

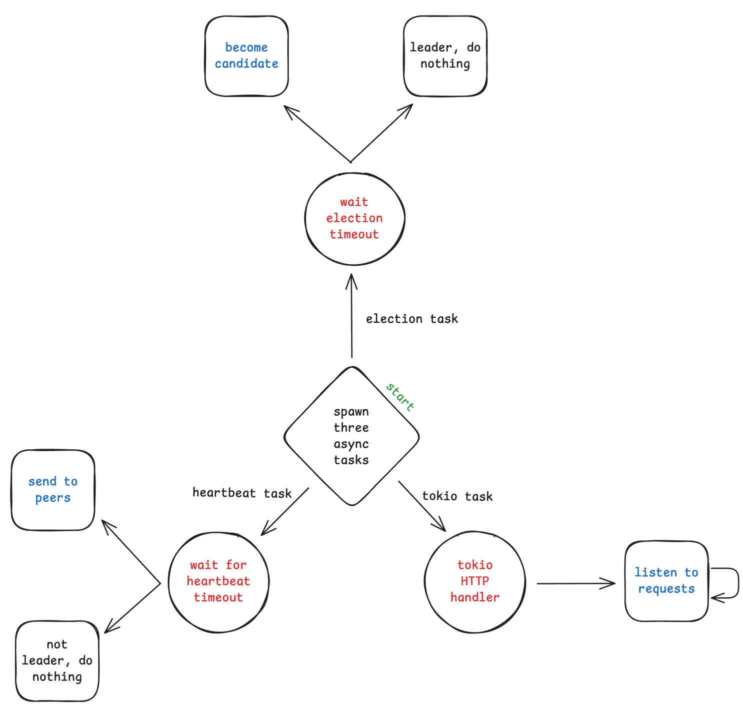


Figure 1: Concurrent task architecture: Election timeout, heartbeat, and HTTP handler tasks operate independently, coordinating through shared state

As shown in Figure 1, the three tasks are:

1. **Election Timeout Task** - Monitors for leader failures and initiates 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<>>`, ensuring thread-safe coordination.

### 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 up-to-date comparison:**

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

### 1.4. Term Management

Terms act as logical clocks. When a node observes a higher term:

- Immediately steps down to follower
- 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).

**Request includes:**

- Leader's current term
- Previous log index and term (for consistency)
- New entries to append
- Leader's commit index

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

### 2.2. Commit Protocol

The leader commits an entry when:
1. A majority of nodes have replicated it
2. The entry is from the current term

```
// 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 |

### 3.2. Storage Format

State is serialized to JSON and written to `./states/raft_state_<id>.json` after every modification:
- Term changes (elections)
- Vote grants
- Log appends

### 3.3. Recovery Process

On startup:
1. Load persistent state from disk
2. Initialize as follower in last known term
3. Wait for leader heartbeat or timeout

## 4. Failure Handling

### 4.1. Leader Failure

**Detection:** Followers detect via missed heartbeats (>400ms).

**Recovery:**
1. Election timeout expires
2. Follower becomes candidate
3. New leader elected within 500ms

**Data safety:** Log entries committed on majority survive leader crashes.

### 4.2. Split Votes

If two candidates start elections simultaneously:
- Both may fail to achieve majority
- Election timeout fires again with **different** random delays
- System eventually converges (typically within 2-3 attempts)

### 4.3. Network Partitions

**Scenario:** Cluster splits into [Leader, A] and [B, C]

The minority partition (Leader, A) cannot commit new entries (no majority). The majority partition (B, C) elects a new leader and continues operating.

When partition heals, the old leader sees higher term and steps down.

## 5. Future Improvements

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