Advanced Software Modelling and Design
Raft implementation as a CTMC

Mert Akpinar
mert.akpinar@studio.unibo.it

Benedetta Pacilli
benedetta.pacilli@studio.unibo.it

June 2025

Contents

Chapter 1

Introduction

This project examines the leader election process in the Raft consensus algorithm. We used continuous-time Markov chain modeling to study its probabilistic behavior. Inspired by ideas from a previous distributed systems course, we wanted to expand our knowledge of consensus algorithms by applying stochastic computing techniques. Raft's structured approach to electing leaders, which includes concurrency, candidate competition, and stabilization methods such as heartbeats, makes it a great fit for CTMC analysis.

We used Scala, PRISM, and Scafi for simulations to model the timing dynamics of leader elections in distributed systems. This approach allowed us to investigate important factors, such as leader stability, system recovery, and vote collisions, in stochastic settings.

Chapter 2

Theoretical Background - RAFT

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems [4]. Raft is similar in many ways to existing consensus algorithms but it has several novel features:

- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Strong Leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand, even though we don't have log replication functionality in our project.

One of the key strengths of Raft is its modular design, which separates different aspects of consensus, such as leader election, log replication, safety, and membership changes, into distinct components. This decomposition enhances understandability by allowing each functionality to be analyzed and implemented independently. In our project, we specifically focused on the leader election mechanism, benefiting directly from this structured approach. By isolating leader election from other Raft functionalities, we were able to model and simulate its behavior using continuous-time Markov chains (CTMC), gaining deeper insights into the probabilistic dynamics of candidate contention, stabilization through heartbeats, and system recovery.

Raft implements consensus by first electing a distinguished leader, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows simply from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

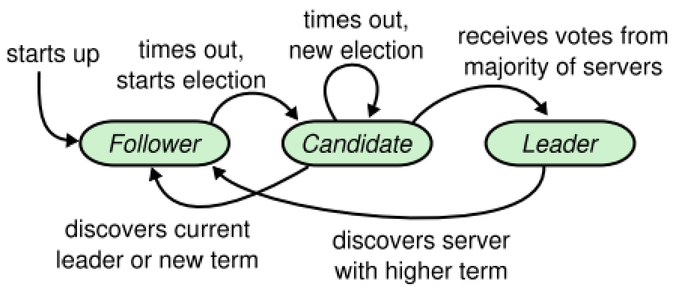
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 2.1: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate who receives votes from most of the full cluster becomes a leader.

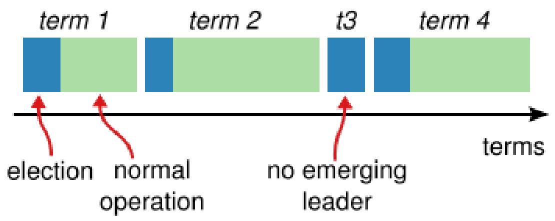

Figure 2.2: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers

A Raft cluster contains several servers; five is a typical number, which allows the system to tolerate two failures. At any given time each server is in one of three states: leader, follower, or candidate. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used to elect a new leader.

Raft divides time into terms of arbitrary length, as shown in Figure 2.2. Terms are numbered with consecutive integers. Each term begins with an election, in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

Terms act as a logical clock [3] in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a current term number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

2.0.1 Leader Election

Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the election timeout, then it assumes there is no viable leader and begins an election to choose a new leader. To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. A candidate continues in this state until one of three things happens: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis. The majority rule ensures that at most one candidate can win the election for a particular term. Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election.

In this project, we concentrate exclusively on the leader election mechanism for simulation purposes. However, the Raft consensus algorithm is designed primarily for ensuring reliable log replication and maintaining communication safety among servers. Beyond leader election, Raft includes essential functionalities such as log replication, log recovery, term-based coordination, commit indexing, and data consistency guarantees. These mechanisms ensure that entries are correctly propagated across nodes, prevent inconsistencies, and allow seamless recovery in case of failures. For a deeper understanding of these key aspects and the full scope of Raft's design, please refer to the official Raft paper [4].

Chapter 3

RAFT as a CTMC

The primary goal of this project was to apply continuous-time Markov chain (CTMC) modeling to a real-world protocol, through the theory and techniques introduced during the course.

Unlike toy examples, Raft's dynamics include both concurrency and contention (e.g., competing candidates), as well as mechanisms for stabilization (e.g., heartbeats), making it a perfect candidate for CTMC-based exploration. Modeling Raft allowed us to simulate a distributed system where *timing is crucial*, allowing us to gain a deeper understanding of how exponential timing distributions influence behavior such as leader stability, system recovery, and vote collisions.

Moreover, since Raft separates logic cleanly into roles and time-driven actions, we were able to map its semantics directly into a structured state machine with well-defined transitions.

3.1 Modelling

In this section, we describe our continuous-time Markov chain (CTMC) model of the Raft leader election mechanism. Transitions are encoded using a custom domain-specific language (DSL), which represents state transitions as labeled actions with associated exponential rates. Based on the DSL, the system evolves as a pure jump process over a finite state space, where every possible state transition is governed by a stochastic rate.

We model not just abstract states, but an explicit stateful view of the full system, including timeouts, vote state, and leadership status, following the Raft paper [4].

3.1.1 System Overview

According to the Raft specification, a node may be in one of three primary states: **Follower**, **Candidate**, or **Leader**, as shown in Figure 3.1. We also added a fourth state, **Crashed**, to enable modeling of failures and recoveries in a distributed setting; these failure/recovery transitions are explicitly modeled with stochastic rates. Figure 3.1 also shows transitions among states, adapted from the original paper. This diagram is used as the basis for our CTMC transition function.

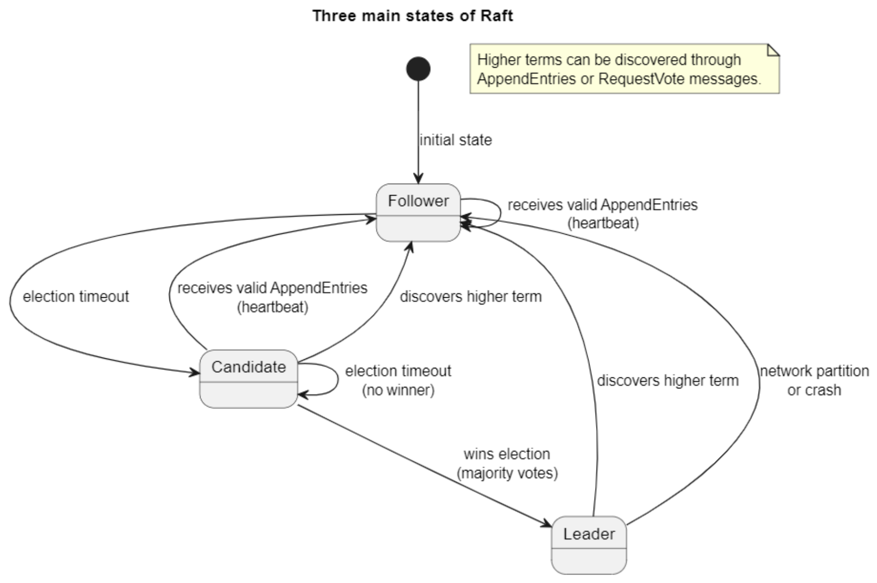
Three main states of Raft
Higher terms can be discovered through AppendEntries or RequestVote messages.
initial state
Follower
receives valid AppendEntries (heartbeat)
election timeout
receives valid AppendEntries (heartbeat)
discovers higher term
Candidate
election timeout (no winner)
discovers higher term
network partition or crash
wins election (majority votes)
Leader

Figure 3.1: RAFT state diagram showing the three main states: Follower, Candidate, and Leader

3.1.2 Model Structure

The system state at each moment is described by an explicit, fully observable snapshot of all servers in the network, represented as a Scala case class:

```
case class ServerState(
  servers: Map[Int, Server],
  votes: Map[Int, Set[Int]],
  currentTerm: Int
)
```

Each server maintains its role, term, vote, and election timeout status, so that all components relevant to transition eligibility are encoded as part of the state/role.

```
case class Server(
   id: Int,
   role: Role,
   term: Int,
   votedFor: Option[Int],
   electionTimeout: Double
)
```

Exponential Transition Rates All temporal dynamics in our model are governed by exponential distributions. Transition rates are either globally constant or locally computed based on the server's state. Table 3.1 lists the key transitions and their associated rates:

Rate	Constant	Event
≈ 3.33	`1/ELECTION_TIMEOUT`	Follower timeout (start election)
Computed per-node	`1/server.electionTimeout`	Individual timeout trigger
50.0	`1/BROADCAST_TIME`	Leader heartbeat (AppendEntries)
0.5	`50/MTBF`	Leader crash
0.2	1/5.0	Recovery from crash

Table 3.1: CTMC transition rates and their corresponding Raft-level behaviors.

This setup allows us to observe realistic dynamics such as spontaneous leader failure and recovery, vote splits under poor timing, and stable leadership under normal operation. It also satisfies timing constraints discussed in the Raft paper [4], specifically:

$$\text{broadcastTime} \ll \text{electionTimeout} \ll \text{MTBF}^{1}$$

This ensures that the probabilistic interleaving of events such as heartbeats, timeouts, and failures produces realistic and analyzable system dynamics.

Timeout Modeling and Symmetry Breaking To ensure followers can initiate elections, we simulate timeout expiration via exponential transitions with *per-node* rates, modeling Raft's use of *randomized election timeouts* to reduce the likelihood of vote splits during leader election. Timeouts are modeled using exponential distributions with randomized parameters in the range $[0.15, 0.3]$, assigned independently to each server at initialization. This allows us to simulate staggered timeout expirations across followers and candidates, which breaks symmetry probabilistically, avoiding deterministic race conditions in a decentralized protocol. If multiple followers time out and become candidates simultaneously, their election campaigns may interfere, leading to repeated failures to elect a leader. By using exponential distributions and randomized parameters for each server, we preserve the memoryless property while still capturing Raft's desynchronization.

[1]Mean Time Between Failures

Vote Collection and Majority Detection Votes are tracked in a map of candidate IDs to sets of voting servers. When a Candidate node receives a vote, it means that the voting follower has not already voted in this term, and the candidate's term is at least as up-to-date. The vote map is then analyzed to detect whether the candidate has achieved a majority. If true, the candidate is promoted to leader.

Heartbeats and Role Stabilization Leaders issue heartbeats to stabilize followers and suppress their timeouts. Heartbeats also serve as a synchronization mechanism: any node (follower, candidate, or stale leader) receiving a heartbeat with a higher term will immediately step down to follower. This logic encodes Raft's leader-driven term propagation and satisfies both the **Leader Completeness** and **Election Safety** properties.

3.1.3 CTMC Definition

All system evolution is expressed through memoryless transitions between well-defined states. Transitions include:

- **Timeout expiration**: a follower or candidate times out and potentially changes state.
- **Vote collection**: a candidate sends stochastic vote requests to eligible voters.
- **Heartbeat propagation**: a leader periodically sends messages, synchronizing terms and resetting timers.
- **Crashes and recoveries**: a leader may crash, and any crashed node may recover independently.

Each of these is implemented as a separate CTMC action labeled with a rate:

```
val raftCTMC: CTMC[ServerState] = CTMC.ofFunction { state =>
  val timeoutTransitions = ...
  val roleTransitions = ...
  val heartbeatTransitions = ...
  val candidateTimeouts = ...
  timeoutTransitions ++ roleTransitions ++ heartbeatTransitions ++ candidateTimeouts
}
```

This form mirrors the CTMC formalism: ($s \xrightarrow{r} s'$), where each transition from state s to s' is annotated with an associated exponential rate r.

3.2 Analysis in Scala

To evaluate the behavior of our CTMC-based Raft model under realistic timing and quorum conditions, we implemented three simulation scenarios. Each scenario highlights a distinct aspect of the system's dynamics: leader election stability, vote contention, and fault tolerance.

3.2.1 Scenarios tested

1. **Scenario 1 - Normal Startup (Odd Number of Nodes):** In this baseline scenario, we simulate a 7-node Raft cluster. All nodes begin as followers, and leader election proceeds according to randomized timeouts. This scenario serves as a control to verify that the system behaves as expected when fault-free, using randomized but fair timeouts to ensure rapid convergence and stable leadership.

2. **Scenario 2 - Split Vote (Even Number of Nodes):** This scenario simulates a 6-node cluster to expose the impact of an even quorum size on Raft's leader election process. The main goal is to detect and quantify occurrences of split votes—situations in which multiple candidates exist but no majority is achieved. During the simulation, we track how many servers are in the *Candidate* role at each step. A split vote is recorded when two or more candidates exist simultaneously and no leader has yet been elected. Formally, for each simulation step, if the number of candidates ≥ 2 and the number of leaders is 0, we increment the `splitVoteCount`. This test illustrates a known limitation for quorum-based consensus algorithms: even-sized clusters are more prone to voting deadlocks unless timeouts are sufficiently randomized to stagger candidacies.

3. **Scenario 3 - Frequent Crashes:** This scenario introduces leader instability by artificially boosting the rate of leader crashes (to 1.0). The recovery rate remains unchanged, leading to frequent elections and increased system churn. Here, the main interest was to measure the system's ability to recover from repeated leadership failures while maintaining election consistency and quorum formation. This scenario highlights Raft's resilience and robustness, confirming that the protocol can maintain liveness and availability even under more aggressive failure conditions.

3.2.2 Results

Each scenario was executed over 30 independent simulation runs, and we collected key metrics including election timing, leader turnover, system stability, and fault recovery.

Figure 3.2 shows the time taken to elect the first leader in each scenario. All scenarios converged to a leader quickly (median < 0.5s), but scenarios with an even number of nodes showed greater variability. Interestingly, the frequent crash scenario also achieves fast leader election times, likely because frequent leadership loss forces elections.

Figure 3.3 confirms that split votes occur most often in the even-numbered cluster. However, the frequent crash scenario also shows a significant number of split vote events, which emerge from overlapping timeouts during repeated re-elections caused by leader churn.

Figure 3.4 indicates that leader stability is highest in the odd and even scenarios, with average tenures above 3 seconds. Under frequent crash conditions, tenure is reduced to under 2 seconds on average, confirming the disruptive effect of rapid failures.

As shown in Figures 3.5 and 3.6, leader crashes and corresponding re-elections are virtually absent in the first two scenarios, but are very prominent in the Frequent Crashes case. Here, the system undergoes over 10 leader failures and up to 50 re-elections per run, highlighting Raft's ability to maintain availability even under high churn.

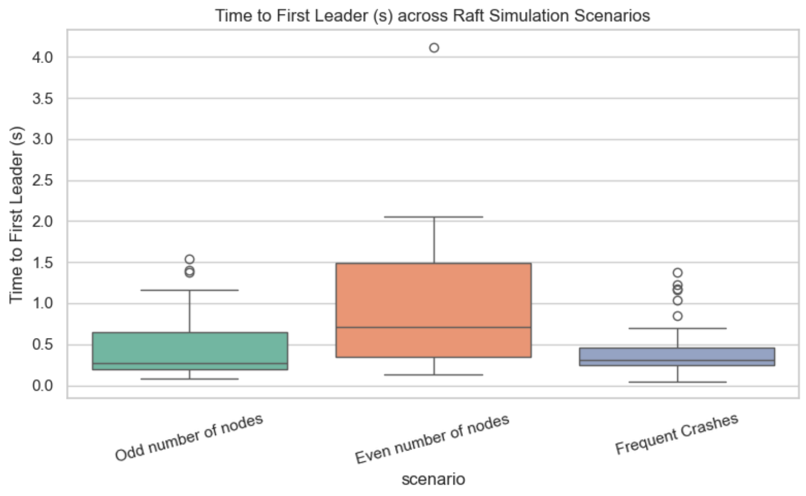

Figure 3.2: Time to First Leader across Raft Simulation Scenarios

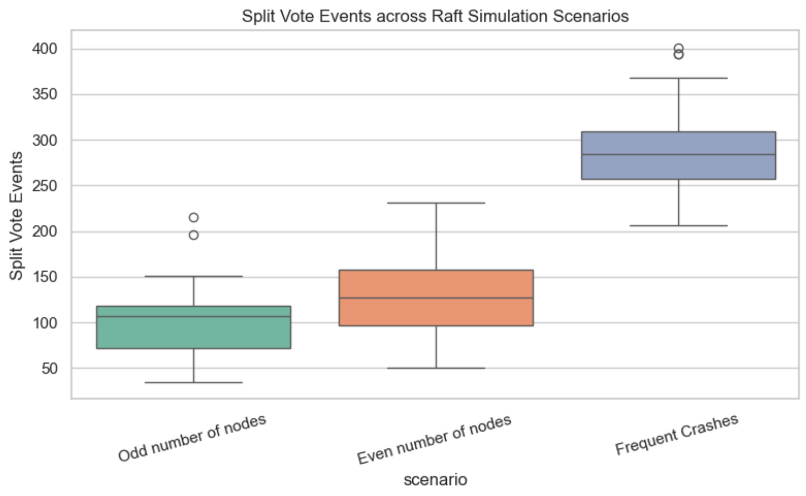

Figure 3.3: Split Vote Events across Raft Simulation Scenarios

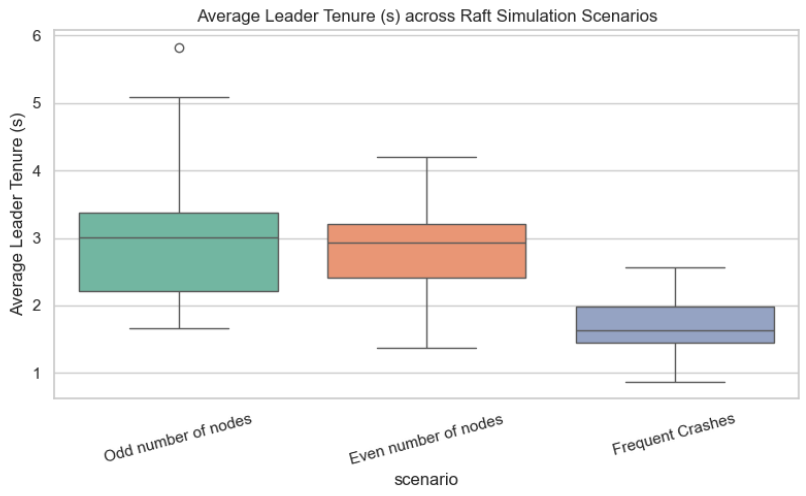

Figure 3.4: Average Leader Tenure across Raft Simulation Scenarios

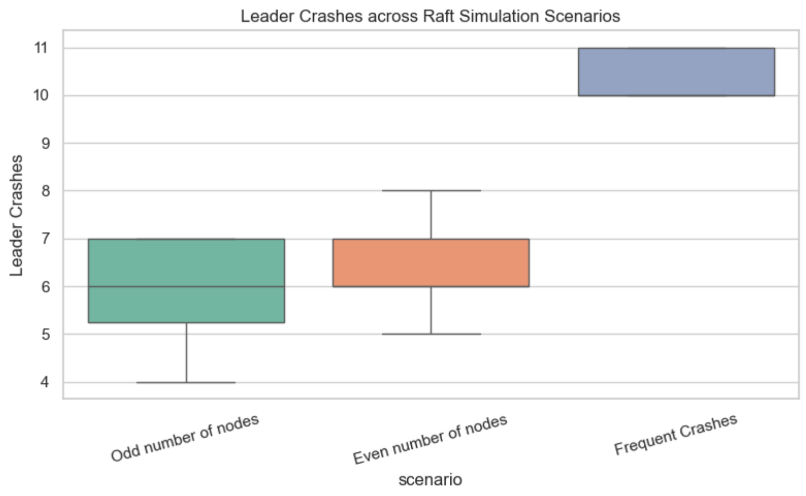

Figure 3.5: Leader Crashes across Raft Simulation Scenarios

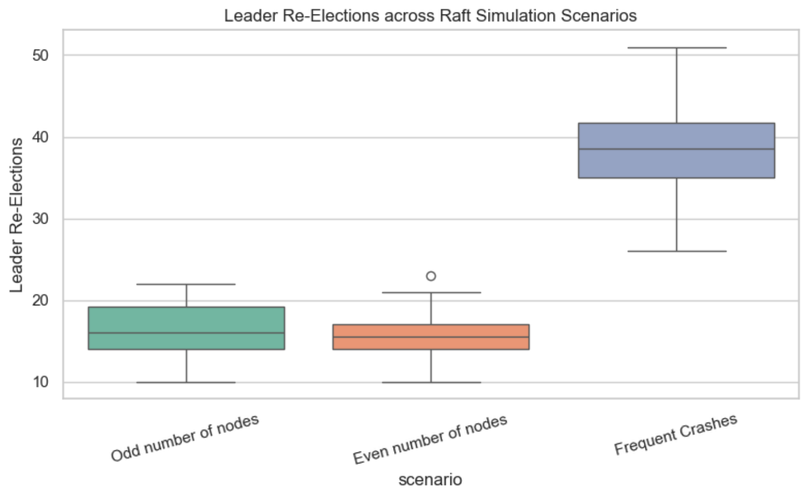

Figure 3.6: Leader Re-Elections across Raft Simulation Scenarios

Chapter 4

Prism

To formally verify key properties of our Raft leader election model, we developed two continuous-time Markov chain (CTMC) models in PRISM [2]: one for an odd-sized cluster (3 nodes) and one for an even-sized cluster (4 nodes). This allowed us to compare convergence behavior and protocol correctness under both quorum-stable and quorum-challenged configurations.

4.1 Modelling

Each server is modeled as an independent module with four possible roles: `Follower`, `Candidate`, `Leader`, and `Crashed`. The servers interact via synchronized transitions for events such as timeouts, voting, leader election, heartbeat propagation, and crash/recovery.

The shared module maintains global state:

- A counter for votes granted in the current term.
- Boolean flags for recent heartbeat receipt.
- Per-server term counters, which track the version of the election cycle each node believes in.

Transitions follow the Raft protocol's rules:

- A follower that times out and hasn't received a heartbeat becomes a candidate.
- Candidates request votes if their term is up-to-date.
- A leader is elected when a candidate obtains a majority of votes (2 of 3 in the odd case, 3 of 4 in the even case).
- Leaders periodically send heartbeats to reset follower timers.
- All roles can crash and recover, maintaining CTMC dynamics.

4.1.1 Rates and Synchronization

The model uses fixed exponential rates:

- `TIMEOUT_RATE` = 3.33: governs when followers become candidates.
- `VOTE_RATE` = 5.0: determines how quickly votes are collected.
- `HEARTBEAT_RATE` = 100.0: simulates frequent heartbeat emission.
- Crash and recovery are modeled as stochastic events with rates 0.01 and 0.2, respectively.

Heartbeat transitions are synchronized across modules using labeled actions (e.g., `[heartbeat1to2]`) to propagate term updates and reset follower timeouts. To simulate timeout decay, a dummy `reset_heartbeats` transition resets all heartbeat flags periodically.

4.1.2 Odd vs. Even Configurations

We implemented two versions of the model:

- **Odd-node (3 servers):** With an easy majority (2 of 3), this configuration tends to converge quickly and consistently.
- **Even-node (4 servers):** Requires 3 of 4 votes, which increases the chance of election contention due to vote splits and term desynchronization.

By keeping the model structure identical across both versions (aside from the number of nodes and voting thresholds), we enable a fair comparison of how node parity affects convergence guarantees and correctness properties.

4.2 Simulation

Using PRISM, we evaluated six key correctness and liveness properties over both the 3-node and 4-node CTMC Raft models.

Leader Uniqueness

- **Property:** No two servers are ever leaders simultaneously.
- **Formalization:**

$$P \leq 0\,[\,\mathsf{F}\,(s_X = 2 \wedge s_Y = 2)\,] \quad \forall X \neq Y$$

- **Result:** `true` for both 3-node and 4-node models.

This confirms Raft's **Election Safety** guarantee: a unique leader is always maintained, regardless of cluster size or timing variations.

Term Consistency (Election Safety)

- **Property:** If a node is leader, its term is at least as large as all other nodes.
- **Formalization:**

$$P \geq 1\,[\,\mathsf{G}\,(\forall X\ \ (s_X = 2 \Rightarrow \forall Y \neq X\ (\mathsf{term}_X \geq \mathsf{term}_Y)))\,]$$

- **Result:** `true` in both models.

This ensures that outdated leaders cannot arise. Every leader's term reflects the most recent known state in the system.

Time-Bounded Liveness

- **Property:** With at least 99% probability, a leader is elected within 3 time units.
- **Formalization:**

$$P \geq 0.99 \left[\mathsf{F}_{\leq 3} \left(\bigvee_X s_X = 2 \right) \right]$$

- **Result:** `true` in both models.

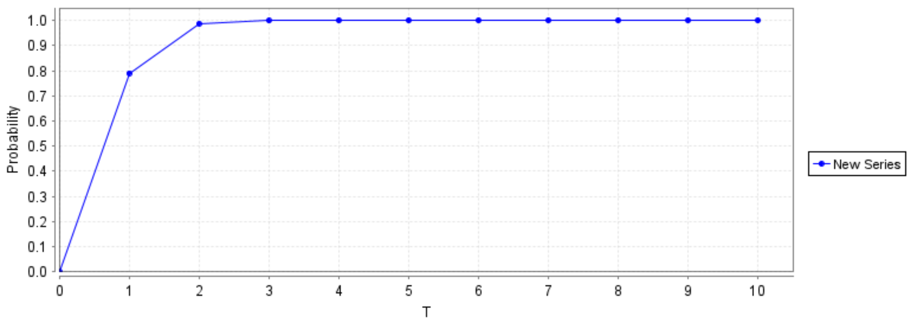

Figure 4.1: Probability of Leader Election by Time T (3-node cluster)

Leader Election Probability over Time

- **Property:** What is the probability that a leader is elected within T time units?
- **Formalization:**

$$P =? \left[\mathsf{F}_{\leq T} \left(\bigvee_X s_X = 2 \right) \right]$$

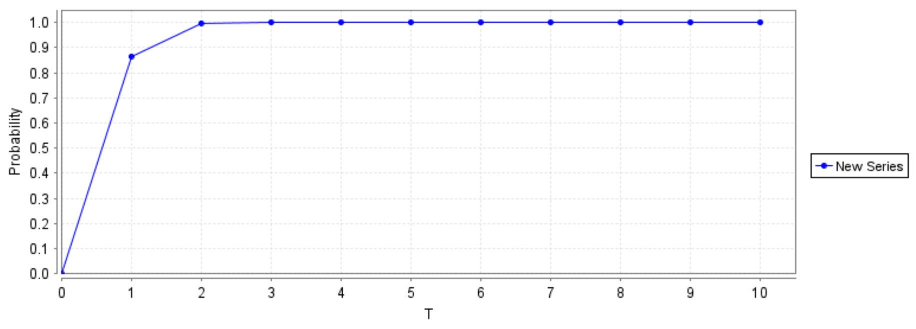

Figure 4.2: Probability of Leader Election by Time T (4-node cluster)

- **Result:** Probability converges to ~ 1.0 by $T = 3$ for both cluster sizes, as shown by Figure 4.1 and Figure 4.2.

Figure 4.2 shows that Raft's CTMC dynamics allow fast convergence to a leader under most conditions, even in the presence of potential vote splits.

Term Divergence

- **Property:** What is the probability that any server's term exceeds another's by more than k?

- **Formalization:**

$$P =? \left[\mathsf{F} \left(\exists X, Y \; \left| \mathsf{term}_X - \mathsf{term}_Y \right| > k \right) \right]$$

We evaluated this property parametrically for term divergence thresholds > 1, > 2, and > 3 across both cluster sizes. The following table reports the probability of reaching such a divergent state at least once during execution.

Max Term Difference	**3 Nodes**	**4 Nodes**
> 1	1.73×10^{-4}	2.36×10^{-4}
> 2	1.68×10^{-7}	1.45×10^{-7}
> 3	1.19×10^{-9}	9.52×10^{-10}

Table 4.1: Probability of exceeding term difference thresholds in Raft CTMC models

These results demonstrate that term divergence is not only rare but also quickly corrected by the protocol. The probabilities drop off sharply as the difference increases, confirming that Raft ensures bounded inconsistencies even under continuous-time dynamics. Large divergences are virtually eliminated by quorum coordination and heartbeat term propagation.

Chapter 5

Raft in Scafi

To gain deeper insights into the leader election mechanism of the Raft consensus algorithm, we utilized ScaFi [1] for simulation. ScaFi provides a visual representation of the system's nodes, allowing us to dynamically observe their state transitions and term updates. This approach enabled us to verify the correctness of the election logic by ensuring that term increments ceased after leader election and that terms stabilized appropriately across nodes. Through ScaFi, we were able to manipulate system conditions to analyze specific behaviors. For instance, we manually forced a leader into the follower state to observe how the system responded to leadership turnover. Additionally, we tested the impact of isolating a node from the network and then reintegrating it, examining how it synchronized its term and role upon rejoining.

5.1 Modeling

Different from previous sections, each server is modeled without the `Crashed` state. Instead of having the `Crashed` state, the sense functionality that gives access to local sensors is used to put the leader into the `Follower` state manually.

In the ScaFi experiments, connected nodes of the system operate by observing the neighboring nodes in the system.

```
val leaderSeenTerm = foldhood(-1)(Math.max) {
    val neighborRole = nbr { roleCode }
    val neighborTerm = nbr { currentTerm }
    if (neighborRole == 2) neighborTerm else -1
}
```

The behavior of individual nodes is designed on top of the state and in terms of other nodes in the system. For instance, in this code block, the node retrieves the maximum term that exists on the cluster and stores it in the following way for future actions.

```
val seesLeader = leaderSeenTerm >= 0
val nextTime = if (seesLeader) 0 else timePassed + 1
```

Each node determines its next term by considering the current states of other nodes:

- If a leader exists, sync with the leader's term.
- If a neighbor with a higher term exists, sync with the neighbor's term.
- If a leader does not exist, and the current time for this round is greater than the timeout, then the node increments its term to become a candidate.
- The node continues with the same term in the default case.

```
val maxNeighborTerm = foldhood(currentTerm)(Math.max)(nbr { currentTerm })
      val nextTerm =
        if (leaderSeenTerm > currentTerm) leaderSeenTerm
        else if (maxNeighborTerm > currentTerm) maxNeighborTerm
        else if (!seesLeader && nextTime > timeout && currentRole != Leader)
            currentTerm + 1
        else currentTerm
```

The voting mechanism is applied by observing candidate neighbors and granting a vote to the first neighbor in the list.

```
val candidateNeighbors = foldhood(Set.empty[(Int, Int)])(_ ++ _) {
        val neighborRole = nbr { roleCode }
        val neighborTerm = nbr { currentTerm }
        val neighborId = nbr { mid() }
        if (neighborRole == 1 && neighborTerm == nextTerm)
            Set((neighborId, neighborTerm)) else Set.empty
      }


      val newVotedFor = currentRole match {
        case Follower if currentVotedFor.isEmpty && candidateNeighbors.nonEmpty =>
          Some(candidateNeighbors.head._1)
        case _ => None
      }
```

The transition rules are the same as in previous chapters:

- Follower translates to Candidate if there is no Leader in the cluster and the timeout has expired.
- Candidate becomes a Leader if more than half of the nodes voted for the Node, goes back to the Follower if a Leader is recognized, or the election time expires.

- Leader goes back to Follower if a node with a higher term has entered the cluster or it is manually forced to go back to the Follower state.

```
case Follower =>
    if (!seesLeader && nextTime > timeout) Candidate else Follower

case Candidate =>
    if (votesReceived > total / 2) Leader
    else if (nextTime > timeout * 4 || seesLeader) Follower
    else Candidate

case Leader =>
    if ((maxNeighborTerm > currentTerm) || forceToFollower) Follower else Leader
```

5.2 Simulation

ScaFi simulation starts with 10 nodes that are connected, forming a cluster. The neighbor range parameter is given as 350px to have all the nodes connected on the initial run.

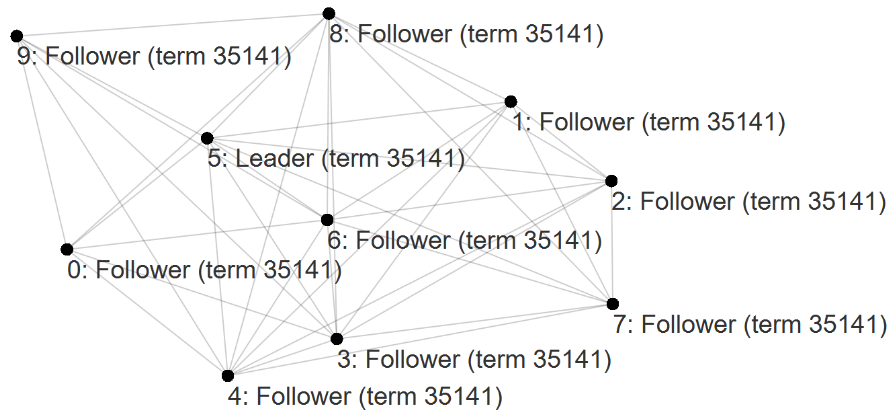

Figure 5.1: Stable state after initial simulation run

Through the simulation, the transitions of nodes between Candidate and Follower states are observable. After the first successful election, all the nodes stop incrementing their terms and sync themselves with the leader's term.

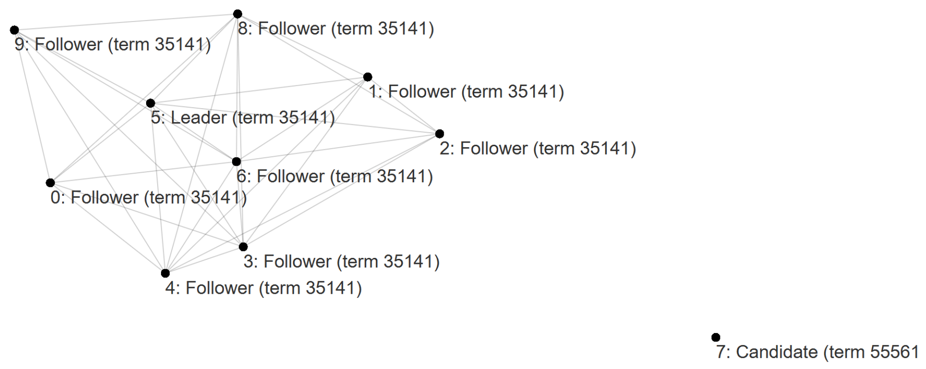

Figure 5.2: Single node leaves the cluster

When a node leaves the cluster, the cluster is still managed by the leader, so there is no change in the terms. On the other hand, since the single node lost the connection with the cluster, it forms its own system and starts to transition between Candidate and Follower, incrementing its term. The reason behind that is the search for a new leader.

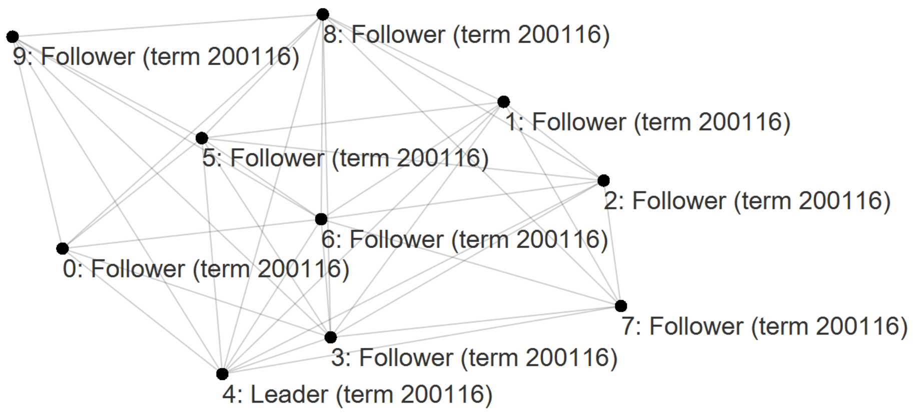
9: Follower (term 200116)
8: Follower (term 200116)
1: Follower (term 200116)
5: Follower (term 200116)
2: Follower (term 200116)
6: Follower (term 200116)
0: Follower (term 200116)
7: Follower (term 200116)
3: Follower (term 200116)
4: Leader (term 200116)

Figure 5.3: Single node re-joins to the cluster

When the node re-joins the cluster with a higher term, the leader steps down as a Follower and the nodes initiate a new election and pick a new leader.

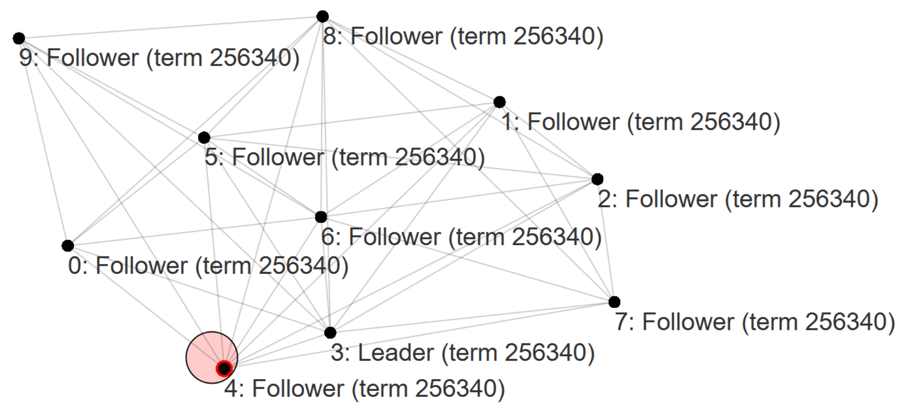
8: Follower (term 256340)
9: Follower (term 256340)
1: Follower (term 256340)
5: Follower (term 256340)
2: Follower (term 256340)
6: Follower (term 256340)
0: Follower (term 256340)
7: Follower (term 256340)
3: Leader (term 256340)
4: Follower (term 256340)

Figure 5.4: Leader's state manually set as Follower

To simulate the leader failure scenario, we manually set the leader's state to follower, observing that the nodes recognize the change in the cluster directly and initiate a new election.

Modeling Raft in ScaFi allowed us to shift from reasoning about individual device behaviors to reasoning at the collective, system-wide level. By implementing leader election through ScaFi primitives like *rep*, *nbr*, and *foldhood*, we gained direct insight into how distributed coordination can emerge from local interactions, even in the presence of failures or partitions. Our simulations also demonstrated the self-stabilising nature of Raft: the system consistently recovered to a stable leader state after disruptions. Ultimately, ScaFi offered a powerful platform for expressing and visualizing consensus dynamics as a composition of space-time computational fields,

Conclusions

This project offered a multifaceted analysis of Raft's leader election mechanism by combining CTMC modeling, formal verification, and aggregate simulation. Modeling Raft as a continuous-time Markov chain allowed us to explore how timing, crashes, and candidate contention influence system behavior. Through simulation scenarios, we observed the effects of quorum size, vote splitting, and failure recovery on leader stability and responsiveness.

Using PRISM, we formally verified key correctness and liveness properties such as leader uniqueness, term consistency, and election convergence.

Our ScaFi implementation highlighted how consensus can emerge from local interactions in a decentralized system. By simulating leader turnover, node isolation, and reintegration, we confirmed Raft's self-stabilising behavior in a spatial, dynamic context.

Bibliography

[1] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming, Nov 2022.

[2] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[3] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.