Modelling and Verification of Raft Consensus Protocol

Shobhit Singh

Chennai Mathematical Institute

April 21, 2025

Overview

What is Raft

What is Distributed Consensus?

- Distributed consensus ensures that multiple nodes in a distributed system agree on a common value despite failures.
- **Challenges:**
  - Network partitions and asynchrony.
  - Node failures and leader crashes.
  - Ensuring consistency while allowing availability.
- **Common use cases:**
  - Replicated state machines (e.g., distributed databases, key-value stores).
  - Coordination services (e.g., Zookeeper, etcd, Consul).

Why Raft Over Paxos?

- Paxos is the traditional consensus algorithm but is considered complex and difficult to understand.
- **Raft was designed for understandability** while maintaining the same correctness guarantees.
- Raft provides a **strong leader model**, making it easier to reason about system behavior.

Introduction to Raft

Raft is a consensus algorithm designed to be:

- **Understandable** – clarity over complexity.
- **Consistent** – all nodes agree on log contents.
- **Fault-tolerant** – tolerates failures of minority nodes.

Introduction to Raft

Raft is a consensus algorithm designed to be:

- **Understandable** – clarity over complexity.
- **Consistent** – all nodes agree on log contents.
- **Fault-tolerant** – tolerates failures of minority nodes.

Its goal is to ensure distributed state machines apply the same sequence of commands, even in the presence of failures.

Introduction to Raft

Raft is a consensus algorithm designed to be:

- **Understandable** – clarity over complexity.
- **Consistent** – all nodes agree on log contents.
- **Fault-tolerant** – tolerates failures of minority nodes.

Its goal is to ensure distributed state machines apply the same sequence of commands, even in the presence of failures.
Doesn't prevent against Byzantine failures. Only server death, network timeouts, message drops etc.

Introduction to Raft

- Raft decomposes the consensus problem into three subproblems:
  1. **Leader Election**: Choosing a single leader to manage the log.
  2. **Log Replication**: Ensuring logs on all servers are consistent.
  3. **Safety**: Maintaining the correctness of the log.
- Every change to the system state goes through the leader.

Raft Server States and Persistent State

- A Raft server can be in one of three states:
  - **Leader**: Handles all client interactions.
  - **Follower**: Passive state, responds to requests.
  - **Candidate**: Temporary state to initiate elections.
- Persistent state on all servers:
  - `currentTerm`, `votedFor`, `log[]`
- Volatile state:
  - `commitIndex`, `lastApplied`

What is a Term?

- Raft divides time into periods called **terms**.
- Each term begins with a new election.
- Terms are numbered with monotonically increasing integers.
- `currentTerm` is the highest term a server has seen.
  - Used to detect stale leaders and RPCs.
- If a server sees a term higher than its current term, it updates its `currentTerm` and becomes a follower.

Triggering an Election

- Followers start an election after election timeout.
- Transition steps:
  - Increment `currentTerm`
  - Become Candidate
  - Vote for self: `votedFor = self`
  - Send `RequestVote` RPCs to all servers

The RequestVote RPC Explained

- RPC format: `RequestVote(term, candidateId, lastLogIndex, lastLogTerm)`
- Vote granted if:
  1. `term` $\geq$ `currentTerm`
  2. Not voted yet in this term
  3. Candidate's log is at least as up-to-date:
     - `lastLogTerm > receiverLastLogTerm` or
     - `lastLogTerm == receiverLastLogTerm` and `lastLogIndex >= receiverLastLogIndex`
- On majority votes (`> N/2`), candidate becomes Leader.

Election Outcomes

- **Win**: Candidate gets majority, becomes Leader.
- **Lose**: If it recieves a any sort of message where the term ¿= currentTerm.
  - Reverts to Follower if recieved term $\geq$ `currentTerm`
- **Split Vote**: No majority due to simultaneous candidates.
  - Timeout and retry with incremented term

Dealing with Split Votes

- Randomized election timeouts reduce split votes:
  - Random wait before transitioning to candidate
  - Reduces collisions, increases chance of one clear winner
- Candidates that fail restart election with new timeout

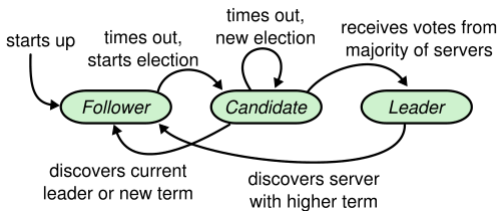
Election Safety Properties

- **Election Safety**: Only one leader elected per term
- Enforced by:
  - One vote per server per term
  - Majority requirement guarantees uniqueness
- No two leaders in same term possible

Timeline of an Election (Example)

1. Follower A times out, becomes Candidate (term = 2)
2. Votes for self, sends RequestVote to B, C, D
3. B and C grant vote (log up-to-date)
4. A gets 3/5 votes -¿ becomes Leader
5. Sends heartbeats (AppendEntries with no entries)
6. Prevents other elections

Summary of Leader Election Process



- Process:
  - Timeout $\rightarrow$ Candidate $\rightarrow$ RequestVote $\rightarrow$ Majority $\rightarrow$ Leader
- Handles split votes with randomized timeouts
- Guarantees:
  - **Election Safety : Unique leader per term**

Techniques Used

Model Checking

- **Model Checking** is an automated technique to verify whether a system satisfies a temporal logic specification (e.g., in LTL or CTL).
- The system is modeled as a transition system:

$$\mathcal{M} = (S, I, T)$$

where:

- S: Set of states
- $I(s)$: Predicate defining initial states
- $T(s, s')$: Transition relation

BMC vs SMC

Two Approaches:

- **Symbolic Model Checking (with BDDs):**
  - Uses Binary Decision Diagrams to represent I, T, and sets of states.
  - Explores the full reachable state space.
  - Can prove properties but may suffer from BDD blowup.
- **Bounded Model Checking (BMC):**
  - Unrolls the system up to depth k and searches for counterexamples.
  - Encodes the problem as a SAT/SMT formula.
  - Efficient for bug finding; incomplete for full correctness.

Bounded Model Checking (BMC): Encoding

- BMC checks if a counterexample exists within k steps.
- Construct a logical formula and query a SAT/SMT solver for satisfiability.

Formula for BMC:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\varphi(s_i)$$

- $I(s_0)$: Initial state predicate
- $T(s_i, s_{i+1})$: Transition relation unrolled up to step k
- $\neg\varphi(s_i)$: Violation of the desired property at any step $i \in [0, k]$

- If satisfiable, a counterexample trace exists.
- If unsatisfiable, no violation exists within bound k.
- Cannot conclude correctness beyond the bound.

Normal Induction vs K-Induction

Goal: Prove property $P(s)$ holds in all reachable states of a system.

Standard Induction:

- **Base Case:** $P(s_0)$
- **Inductive Step:**
  $P(s) \wedge T(s, s') \Rightarrow P(s')$

K-Induction:

- **Base Cases:** $P(s_0), \ldots, P(s_k)$
- **Inductive Step:**

$$P(s_0), \ldots, P(s_{k-1}),$$
$$T(s_0, s_1), \ldots, T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

Why K-Induction is Better

- Can prove properties that are not 1-inductive (e.g., need history)
- More robust for:
  - Loops with delayed effects
  - Finite state machines with memory
  - Software/hardware systems with non-trivial control flow
- Often works where normal induction fails

Why K-Induction is Better

- Can prove properties that are not 1-inductive (e.g., need history)
- More robust for:
  - Loops with delayed effects
  - Finite state machines with memory
  - Software/hardware systems with non-trivial control flow
- Often works where normal induction fails

Trade-off: More expensive (larger formulas)

Over-Approximation in K-Induction

- Inductive step assumes:

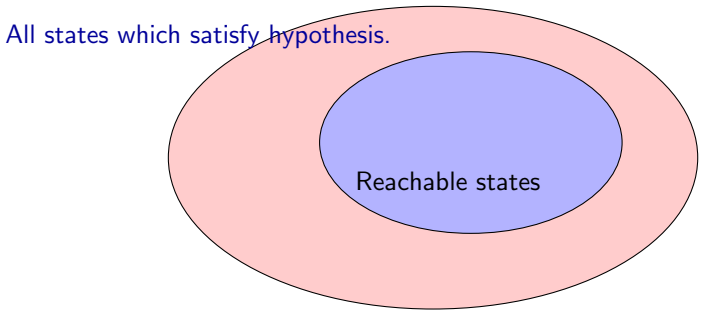
$$P(s_0), \ldots, P(s_{k-1}) \wedge T(s_0, s_1), \ldots, T(s_{k-1}, s_k)$$

- But no requirement that s_0 is reachable!
- So it proves the property over a **superset** of reachable traces.

Implication

If the property holds on this over-approximation,
then it must hold on the actual reachable states.

Overapproximation



If P holds in pink, it holds in reachable! but if its fails, we might need to strengthen the invariant i.e. make it the over-approximation tighter.

Formula Sent to SAT/SMT Solver

For a safety property P, initial predicate I, and transition T:
Base Case (Unrolling to depth k):

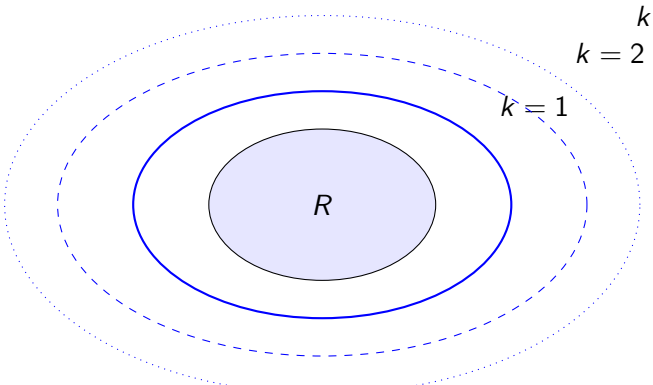
$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \Rightarrow P(s_0) \wedge \cdots \wedge P(s_k)$$

Inductive Step:

$$P(s_0) \wedge \cdots \wedge P(s_{k-1}) \wedge$$
$$T(s_0, s_1) \wedge \cdots \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

Effect of Increasing k

- K-induction checks property $P(s)$ over all k-step execution paths.
- Larger k leads to:
  - Stronger inductive hypothesis
  - Tighter approximation of reachable states
  - More spurious counterexamples eliminated



Over-approximation gets smaller

Objectives

- Come up with a model for Leader Election Protocol and prove the correctness using K-Induction

- Inject the D-duplication (recounting of votes from a server) bug in the model and try to catch it efficiently using Bounded Model Checking.

Models with Different Levels of Abstraction

What is a reasonable abstract model to use that allows invariants proved for an abstract model to expedite BMC at a concrete implementation level?

Took about 50% of the time.

Trivial Model: Level 1 Abstraction

Key Idea:

- A highly nondeterministic abstraction of Raft.
- Simplifies reasoning about safety properties.
- Provides a formal refinement of more concrete implementations.

State Definition:

$$\begin{aligned} \text{Role} &= \{F, C, L\} \quad \text{(Follower, Candidate, Leader)} \\ \text{Procid} &= \{0, 1, \ldots, \text{Max}\} \\ \text{Term} &= \text{Positive Integers} \\ \text{State} &= \text{Node: [Procid]} \ \to \langle R : \text{Role}, t : \text{Term} \rangle \end{aligned}$$

Initial State:

$$\forall i \in \text{Procid}, \quad s[i].R = F \text{ and } s[i].t = 1$$

Trivial Model: Transition Relation

Transition Definition:

$$T(s, s') = \big(\forall i, j \in \text{Procid}, \quad (s'[i].R = L \text{ and } s[j].R = L) \Rightarrow i = j\big)$$

$$\text{and} \quad \big(\forall i \in \text{Procid}, \quad s[i].t \leq s'[i].t\big)$$

Intuition:

- Ensures only one leader exists at any time.
- Time (term number) is non-decreasing for every process.
- Extremely nondeterministic – allows arbitrary transitions as long as these constraints hold.

Leader Election Property (LEP)

LEP Safety Property:

$$LEP(s) = \forall i, j \in \text{Procid}, \quad i \neq j \Rightarrow \neg(s[i].R = L \text{ and } s[j].R = L)$$

Why is this important?

- Ensures at most one leader exists at any time.
- This property trivially holds in our abstraction.
- Any implementation that refines this model must also satisfy LEP.

Refinement and Existential Abstraction

Refinement Definition: A LEP-correct Raft protocol is a **formal refinement** of this model.

Existential Abstraction:

- A concrete model CS refines an abstract model AS if:

$$\forall cs_0, cs_1, \ldots, cs_i, cs_{i+1}, \quad T^{\wedge}(cs_i, cs_{i+1}) \Rightarrow T(A(cs_i), A(cs_{i+1}))$$

- That is, every feasible trace of the concrete system must be mapped to a feasible trace of the abstract model.

Refinement Conditions

Conditions for Existential Abstraction:

- **Initialization Condition (InitCond):**

$$\forall cs \in \mathrm{CS}, \quad \mathrm{Init}(A(cs))$$

- **Simulation Condition (SimulateCond):**

$$\forall i \geq 0, \quad (T^{\wedge}(cs_i, cs_{i+1})) \Rightarrow (T(A(cs_i), A(cs_{i+1})))$$

- **(Conformance Condition (ConservCond):**

$$\forall cs, cs' \in \mathrm{CS}, \quad T^{\wedge}(cs, cs') \Rightarrow T(A(cs), A(cs'))$$

Why does this matter?

- If these conditions hold, all safety properties proven in the abstract model apply to the concrete system.
- Enables efficient verification via bounded model checking.

Criteria for an Abstract Model

To be useful for verifying a safety property, an abstract model should satisfy:

1. **Sufficient State Information:**
   Must capture enough state to exhibit safety violations and support inductive invariants ensuring the property.
2. **Existential Abstraction:**
   Should not exclude any behavior legally permitted by a correct implementation. Can use non-determinism to stay abstract and small.
3. **Non-Inductive Safety Property:**
   Preferably, the safety property itself is not inductive. Otherwise, no supporting invariants would be needed.
4. **Simplicity:**
   The model should be small enough to ease the verification effort.

A Level-2 Abstraction

- Abstracts away network and messages
- Retains roles, term, votes to enable safety proofs
- Goal: Prove Leader Election Property (LEP)

Model Components:

- `Role = {F, C, L}` (Follower, Candidate, Leader)
- `Procid = {0,1,..,Max}`
- `Term = PosInt, Quorum = bool`
- `Votes: [Procid]` $\rightarrow$ `Nat` tracks votes received from each node
- `State: Node = [Procid]` $\rightarrow \langle$`Role, Term, ms: Votes, q: Quorum`$\rangle$

Defining LEP and Invariants

Leader Election Property (LEP):

$$\forall i \neq j : \quad \neg(s[i].R = L \wedge s[j].R = L)$$
$$s[i].R = L \Rightarrow \texttt{ucard}(\{j \mid s[i].ms[j] > 0\}) \geq \texttt{Max}/2 + 1$$

Supporting Invariants:

- **UniqueQ**: At most one node has quorum

$$\forall i \neq j : \neg(s[i].q \wedge s[j].q)$$

- **MajQ**: Quorum only if node received majority

$$s[i].q \Leftrightarrow \texttt{ucard}(\{j \mid s[i].ms[j] > 0\}) \geq \texttt{Max}/2 + 1$$

- **UniqueVote**: No double voting in a term

$$s[i].ms[p] > 0 \wedge s[j].ms[p] > 0 \Rightarrow i = j$$

Transition System: T(s, s')

Initial State:

$$\forall i : \quad s[i].R = F,\ s[i].t = 1,\ \forall j :\ s[i].ms[j] = 0$$

Transitions:

$$\forall i :\ s[i].t \leq s'[i].t \qquad \text{(Term never decreases)}$$

$$s[i].R = C \land \neg s[i].q \land s'[i].q \Rightarrow s'[i].R = L \qquad \text{(Promotion to Leader)}$$

$$s'[i].q \iff \left|\{j \mid s'[i].ms[j] > 0\}\right| \geq \left\lfloor \frac{\text{Max}}{2} \right\rfloor + 1 \qquad \text{(Quorum rule)}$$

$$\forall j \in \text{Procid} :\ s[i].ms[j] \leq s'[i].ms[j] \qquad \text{(Vote monotonicity)}$$

$$\forall j, k \in \text{Procid} :\ (s'[j].ms[i] > 0 \land s'[k].ms[i] > 0) \Rightarrow j = k \qquad \text{(Unique Vote)}$$

Invariants built into transition:

- $\texttt{UniqueQ}(s')$
- $\texttt{MajQ}(s')$
- $\texttt{UniqueVote}(s')$

LEP-Safe Abstraction for Verification

- Abstraction function `A(cs)` maps concrete state to abstract state
- Concrete transition: $T_{cs}(cs_0, cs_1)$
- BMC Check:

$$T_{cs}(cs_0, cs_1) \wedge \cdots \wedge \neg(LEP(A(cs)) \wedge UniqueQ \wedge MajQ \wedge \dots)$$

- Ensures abstraction preserves all allowed behaviors
- Guarantees no two leaders in one term with quorum

Using Abstraction to Verify LEP

Verification via BMC:

$$T_{cs}(cs_0, cs_1) \wedge \cdots \wedge T_{cs}(cs_k, cs_{k+1})$$
$$\Rightarrow \neg[LEP(A(cs)) \wedge UniqueQ(A(cs)) \wedge MajQ(A(cs)) \wedge UniqueVote(A(cs))]$$

Advantage: Catch violations of LEP early in BMC using abstract inductive invariants.

Alternate Strategy: Use more refined models (closer to implementation) to discover deeper inductive invariants, enabling better debugging.

Abstraction Function: Level 2 to Level 1

State Mapping:

$$A(\text{State\_L2}) = \langle R, t \rangle \quad \text{(Role, Term)}$$

where ‘Votes‘ and ‘Quorum‘ are abstracted away.

Transition Mapping:

$$A(T_{\text{L2}}) = T_{\text{L1}} \quad \text{(Only Role and Term transitions considered)}$$

Key Property Preserved:

$$A(\text{LEP}) \quad \text{(LEP is preserved)}$$

Level 3 : Concrete Model

- We model Raft as a distributed system with:
  - Nodes: $\mathcal{N} = \{n_1, \ldots, n_k\}$
  - Message queue (network module): $\mathcal{M}$
- Messages: $m = \langle \text{type}, \text{payload}, s, t_s, r \rangle$
  - s = sender, t_s = sender's term, r = receiver
- All message sending and delivery are nondeterministic
- Network may delay messages arbitrarily but is fair

Node Local State

Each node $n \in \mathcal{N}$ maintains:

$$\begin{aligned}\text{state}_n = \langle & \text{term}_n \in \mathbb{N}, \\ & \text{role}_n \in \{\text{Follower}, \text{Candidate}, \text{Leader}\}, \\ & \text{log}_n : \text{List of } (\text{index}, \text{term}, \text{command}), \\ & \text{votedFor}_n \in \mathcal{N} \cup \{\perp\}, \\ & \text{timeout}_n \in \mathbb{N}, \\ & \text{Votes Received}_n \in \mathbb{N}, \\ & \text{inbox}_n \subseteq \mathcal{M}\rangle\end{aligned}$$

- Node behavior is defined by transition rules reacting to inbox messages and timeouts

Network Module

- Message queue: $\mathcal{M} \subseteq \mathcal{T} \times \mathcal{P} \times \mathcal{N} \times \mathbb{N} \times \mathcal{N}$
- At each step, a message $m \in \mathcal{M}$ may:
  - Be delivered to recipient r, appended to inbox_r
  - Remain in queue (delayed)
- Nondeterministic delivery; fairness ensures eventual delivery

Transition Rules (1): Election

Election Timeout:

- If $\text{role}_n = \text{Follower}$ and $\text{timeout}_n = 0$
- Then:
  - $\text{term}_n := \text{term}_n + 1$, $\text{role}_n := \text{Candidate}$
  - $\text{votedFor}_n := n$
  - For all $r \neq n$, enqueue:

$$\langle \text{RequestVote}, \text{payload}, n, \text{term}_n, r \rangle \in \mathcal{M}$$

Receive RequestVote:

- If $t_s > \text{term}_r$, update term and become Follower
- Grant vote if log is up-to-date and not already voted

Transition Rules (2): Leader Election

Become Leader:

- Candidate c receives majority of VoteResponse(granted=True)
- Then: $\text{role}_c := \text{Leader}$
- Broadcast empty AppendEntries to all $r \neq c$

Receiving AppendEntries:

- If $t_s \geq \text{term}_r$, accept, update term and become Follower
- Else, reject

Model 4: Yet Another Model

- No message sending step; instead, we model possible inbox messages nondeterministically.
- System state includes a set of nodes, each with its local state (term, role, etc.).
- At each step, a node n' is chosen nondeterministically.
- We nondeterministically choose an allowed message m consistent with system state.

Node State

Each node n has:

- $n.\text{term} \in \mathbb{N}$: Current term
- $n.\text{role} \in \{\text{Follower}, \text{Candidate}, \text{Leader}\}$
- $n.\text{votedFor} \in \mathbb{N} \cup \{\perp\}$
- $n.\text{log} = [\text{entries}]$ (omitted for leader election modeling)

Global set: $\text{Terms} = \{n.\text{term} \mid n \in \text{Nodes}\}$

Message Schema

Messages are tuples (M, sender, t) where:

- $M \in \{\text{Hb}, \text{Vr}, \text{Vote}\}$ (Heartbeat, VoteRequest, VoteGrant)
- t is the sender's term

No queueing or delivery delays – messages exist virtually and must be justified by current state.

Allowed Messages by Role (1)

Role	Message	Sender Role	FOL Condition
Follower	Heartbeat	Leader	$\text{sender.role} = L \land t \leq \max(\text{Terms})$
Follower	VoteRequest	Candidate	$\text{sender.role} = C \land t \leq \max(\text{Terms})$
Follower	VoteGrant	Follower	$\text{sender.role} = F \land t < \text{term}_{\text{curr}}$

Allowed Messages by Role (2)

Role	Message	Sender Role	FOL Condition
Candidate	Heartbeat	Leader	$\text{sender.role} = L \land t \leq \max(\text{Terms})$
Candidate	VoteGrant	Follower	$\text{sender.role} = F \land t \leq \text{term}_{\text{curr}}$
Candidate	VoteRequest	Candidate	$\text{sender.role} = C \land t \leq \max(\text{Terms})$
Leader	VoteGrant	Follower	$\text{sender.role} = F \land t \leq \text{term}_{\text{curr}}$
Leader	VoteRequest	Candidate	$\text{sender.role} = C \land t \leq \max(\text{Terms})$
Leader	Heartbeat	Leader	$\text{sender.role} = L \land t \leq \max(\text{Terms})$