Raft Consensus Algorithm Implemented in Distributed Key Value Store

19CSE312 - Distributed Systems

Submitted by:

Name	**Roll Number**
Lokesh Budda Sankar Narayan	CB.EN.U4CSE22026
Sunduru Sai Keerthana	CB.EN.U4CSE22050
Nishnath	CB.EN.U4CSE22023

March, 2025

Contents

1 Problem Statement

Distributed systems require an efficient coordination mechanism to ensure consistency, reliability, and fault tolerance. The study focuses on implementing the Raft consensus algorithm in a distributed key-value store to achieve strong consistency and leader-based coordination.

2 objective

1. Leader Election
2. Log Replication
3. Fault Tolerance
4. Data Consistency
5. Scalability

3 Introduction

Distributed systems consist of multiple independent nodes that must work together while maintaining data consistency across all nodes. A key challenge in distributed environments is ensuring that all nodes agree on the same state, even in the presence of network partitions, node failures, or communication delays. Consensus algorithms provide a structured solution to this challenge by enabling nodes to reach agreement on a shared state.

The **Raft consensus algorithm**, introduced by Diego Ongaro and John Ousterhout in 2014, is designed to be **simple, efficient, and fault-tolerant**, making it a popular choice for **distributed databases and key-value stores**. Unlike its predecessor Paxos, Raft emphasizes understandability without sacrificing performance. This report explores the role of Raft in distributed systems, its operational mechanics, and why it is essential for ensuring reliability and scalability in modern distributed architectures.

4 Distributed Key-Value Store

A **distributed key-value store** is a NoSQL database model that stores data as key-value pairs across multiple nodes, ensuring high availability, scalability, and low-latency access.

4.1 Characteristics

- **Scalability:** Handles increasing loads by adding more nodes horizontally.
- **Fault Tolerance:** Ensures data availability even if nodes fail, using replication.
- **High Performance:** Optimized for fast reads and writes with minimal overhead.
- **Decentralized Architecture:** Reduces single points of failure by distributing control.

4.2 Examples

- **Redis:** An in-memory key-value store used for caching, session management, and real-time analytics.
- **Amazon DynamoDB:** A fully managed cloud-based key-value database with built-in replication.
- **TiKV:** A Raft-based distributed key-value store designed for large-scale transactional workloads.

4.3 Applications

Distributed key-value stores are widely used in modern applications:

- **Web Applications:** For caching and session storage (e.g., Redis in e-commerce platforms).
- **Big Data Processing:** As a foundation for distributed frameworks like Apache Cassandra.
- **Cloud Services:** Supporting scalable infrastructure in AWS and Google Cloud.

5 Raft Consensus Algorithm

Raft ensures consistency across distributed nodes by following a **leader-based approach**. It breaks down the consensus problem into manageable subproblems, making it easier to implement and to understand, compared to other algorithms like Paxos.

5.1 Advantages of Raft

- **Simplicity:** Raft's design is easier to teach and implement than Paxos.
- **Robustness:** Handles failures gracefully with minimal downtime.
- **Extensibility:** Supports cluster membership changes and log compaction.

6 Algorithms in raft.go

6.1 Leader Election

Purpose: Elects a single leader to coordinate the cluster.

Key Steps:

- Nodes wait for `electionTimeout` (randomized 150-300ms).
- If no heartbeat, become `Candidate`, increment `currentTerm`, vote for self.
- Send `RequestVoteArgs` to peers; become `Leader` with majority votes.

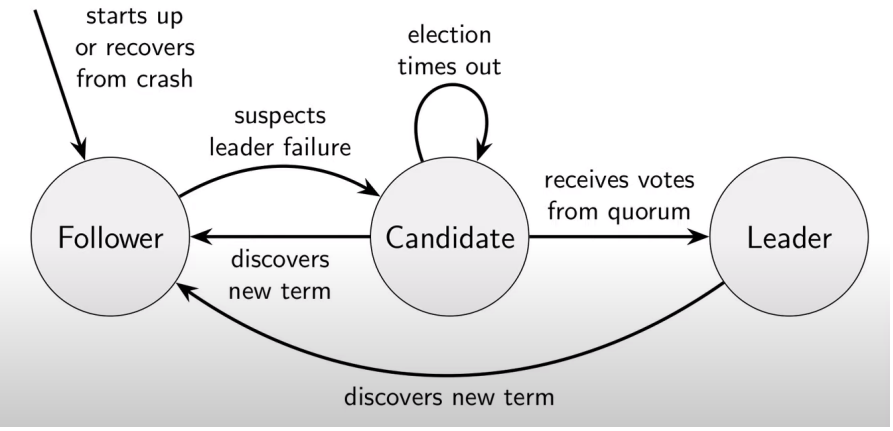

Figure 1: This image represents the Raft consensus algorithm's leader election process. It shows how nodes transition between Follower, Candidate, and Leader states based on timeouts, leader failures, and vote counts.

Code Highlights:

- `Run`: Triggers `StartElection` if timeout exceeded.
- `StartElection`: Increments term, requests votes, transitions to `Leader` on majority.

- `RequestVote`: Grants vote if term is valid and log is up-to-date.

Safety: Higher terms override, ensuring one leader per term.

6.2 Log Replication

Purpose: Replicates leader's log to followers for consistency.

Key Steps:

- Leader sends `AppendEntries` with `PrevLogIndex` and `PrevLogTerm`.
- Followers append entries if prior log matches; leader retries earlier indices on failure.
- Leader commits entries replicated by majority.

Code Highlights:

- `replicateLog`: Sends entries starting at `nextIndex[peer]`, updates indices.
- `AppendEntries`: Appends entries if consistent, updates `commitIndex`.
- `updateCommitIndex`: Advances `commitIndex` with majority replication.

Safety: Log matching prevents inconsistencies.

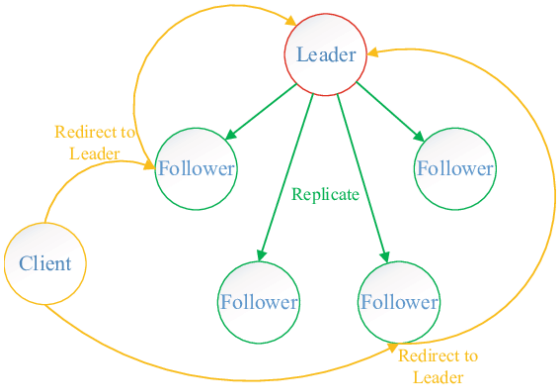

Figure 2: This image represents log replication in the Raft consensus algorithm. The Leader receives client requests and replicates logs to Followers, while Followers forward client requests to the Leader.

6.3 State Application

Purpose: Applies committed log entries to the KV store.
Key Steps:

- Apply entries from `lastApplied` to `commitIndex`.
- Update `kvStore` with `put` commands.

Code: `applyLog`: Iterates log, applies `put` operations, updates `lastApplied`.
Safety: Only committed entries are applied.

6.4 Persistence

Purpose: Ensures state survives crashes.
Key Steps:

- Save `currentTerm`, `votedFor`, `log` to `state.json`.
- Load on startup, apply unapplied entries.

Code:

- `persist`: Writes `PersistedState` to disk.
- `loadPersistedState`: Reads state, resets `commitIndex`/`lastApplied`, applies log.

Safety: Durable state prevents loss of consensus.

6.5 Safety

Purpose: Guarantees one leader per term and log consistency.
Key Steps:

- Term checks ensure newer terms override.
- Log matching rejects inconsistent entries.
- Majority rule for commits ensures durability.

Code: Embedded in `RequestVote`, `AppendEntries`, `updateCommitIndex`.

7 Why Consensus is Needed in Key-Value Stores?

Consensus mechanisms like Raft are crucial for distributed key-value stores because they address fundamental challenges in distributed computing:

- **Consistency:** Ensures all nodes agree on the same data state, avoiding conflicts.
- **Fault Tolerance:** Handles node failures without losing data or compromising availability.
- **Leader Coordination:** Manages updates efficiently in distributed environments by centralizing control through a leader.
- **Conflict Resolution:** Resolves discrepancies that arise from concurrent updates or network delays.

Without a consensus algorithm, distributed key-value stores would face **data conflicts, inconsistencies, and failures in synchronization**. For example, in a leaderless system, concurrent writes to the same key could result in irreconcilable versions of the data. Raft ensures smooth operation by providing a structured mechanism for agreement, making it indispensable for reliable distributed systems.

8 Design Considerations for Distributed Key-Value Stores

Designing a distributed key-value store involves several architectural decisions:

8.1 Data Partitioning

- **Hashing:** Keys are hashed to determine their storage node (e.g., consistent hashing in DynamoDB).
- **Range Partitioning:** Keys are divided into ranges and assigned to nodes (e.g., TiKV).

8.2 Replication Strategy

- **Primary-Backup:** A leader replicates data to followers (used in Raft).
- **Quorum-Based:** Requires a majority of nodes to agree on reads and writes (e.g., Dynamo).

8.3 Consistency Models

- **Strong Consistency:** All nodes see the same data at the same time (Raft's approach).
- **Eventual Consistency:** Nodes converge to the same state over time (e.g., DynamoDB).

9 Performance Evaluation

To evaluate the implementation:

- **Latency:** Measure the time taken for read/write operations under varying loads.

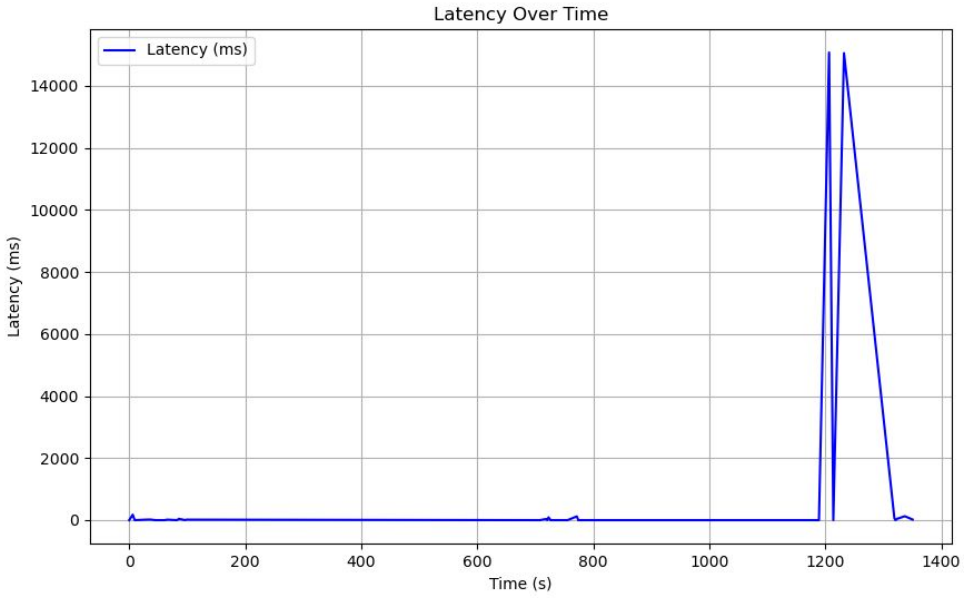

Figure 3: **Latency:** Mostly low, but significant spikes (up to 15 seconds) during replication, likely due to network delays or timeouts.

- **Throughput:** Assess the number of operations per second with increasing node count.

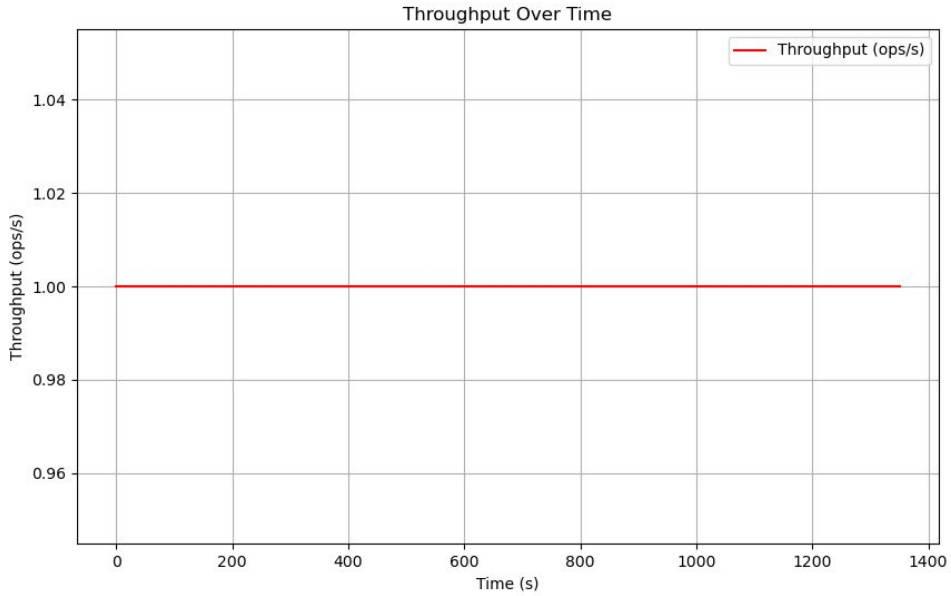

Figure 4: **Throughput:**Very low (1 ops/s) due to manual testing. Needs higher load to evaluate scalability.

- **Fault Tolerance:** Simulate node failures and measure recovery time using Raft's election mechanism.

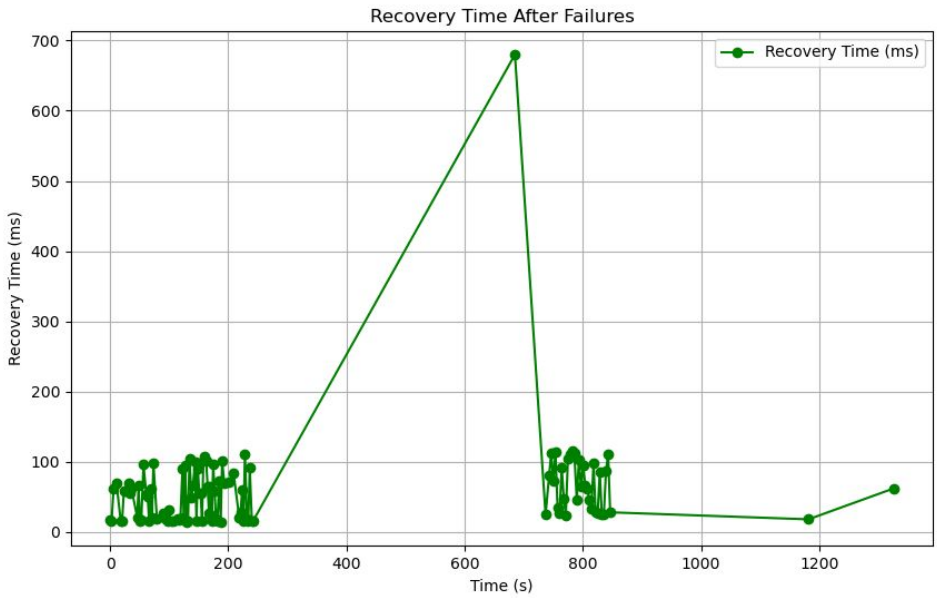

Figure 5: **Fault Tolerance:** Generally fast (50–150 ms), with occasional delays (700 ms) due to election timeouts or network issues.

10 Challenges

Implementing Raft in a distributed key-value store poses challenges:

- **Network Latency:** Delays in message passing can affect election and replication.
- **Scalability Limits:** Large clusters can strain the leader's replication capacity.
- **Debugging:** Distributed systems are notoriously difficult to test and debug.
- **Connecting:** Connecting 2 computers (one as server and another as client) is hard because the persist function is trying to acquire the lock.

11 Conclusion

The **Raft consensus algorithm** is an essential component in distributed key-value stores, ensuring strong consistency, fault tolerance, and efficient coordination. By implementing Raft in Java with concurrency features, developers can

build robust distributed systems capable of handling failures, maintaining synchronized data, and scaling seamlessly. As distributed computing continues to evolve, **consensus algorithms like Raft remain vital for ensuring data integrity and reliability across large-scale infrastructures**.