Raft-Based Key-Value Store in Go

I212481 - Abdullah Zahoor
I210640 - Muneel Haider
CS4049 - BlockChain and Distributed Systems

March 25, 2025

Contents

1 Introduction

Distributed systems must maintain consistency across replicated data while tolerating node failures. Raft is a consensus algorithm designed to be understandable and practical for such systems. In this project, we implement a fault-tolerant, Raft-based distributed key-value store in Go. The system demonstrates core Raft features: leader election, log replication, and state machine application.

2 Objective

The goal of this assignment was to develop a simplified but functional implementation of the Raft consensus protocol using Go. The implementation includes:

- A cluster of Raft nodes that elect a leader among themselves.
- Replication of client commands (`Put`, `Append`, `Get`) across all nodes.
- Reliable communication using Go's net/rpc and synchronization using mutexes.
- A client interface that discovers the leader and interacts with it.

3 System Architecture

3.1 Programming Language and Tools

- **Language:** Go (Golang)
- **Concurrency:** Goroutines and Mutexes
- **Communication:** net/rpc over TCP
- **Build:** go mod and manual compilation

3.2 Module Breakdown

- `raft/node.go`: Implements Raft node state, election logic, leader behaviors.
- `raft/server.go`: Exposes Raft functions as RPC services.
- `raft/raft.go`: Contains data structures for commands and replies.
- `node/main.go`: Boots individual Raft nodes.
- `client/main.go`: Provides a client interface to interact with the leader.

4 Key Functionalities

4.1 Leader Election

Each Raft node starts as a follower. If it does not hear from a leader within a timeout, it becomes a candidate and initiates a vote. A node receiving votes from the majority becomes the leader. Election timeouts are randomized between 5–6 seconds to reduce split vote chances.

4.2 Log Replication

Leaders handle all client commands. Commands are appended to the leader's log and sent to followers through AppendEntries RPCs. Once a majority acknowledges, the command is committed and applied to the state machine (key-value store).

4.3 Command Handling

- **Put:** Sets a key to a value.
- **Append:** Appends a value to an existing key.
- **Get:** Returns the current value of a key (without replication).

4.4 Reliable Startup

To ensure a stable leader election on startup, nodes are given a fixed delay (5 seconds) before they initialize election timers. This prevents premature elections before all nodes are online.

5 Implementation Details

5.1 RPC Interfaces

Raft nodes expose methods over RPC:

- `RequestVote(RequestVoteArgs, RequestVoteReply)`
- `AppendEntries(AppendEntriesArgs, AppendEntriesReply)`
- `StartCommand(Command, CommandReply)`

5.2 Client Discovery

The client sends a dummy command (`Get leader`$_{p}$*robe)toallnodes.Theonethatrespondswithavalidlogindexiscons*

5.3 Command Application

`Commands are only executed once committed. ApplyCommittedEntries() checks and applies entries to the local key-value store. Get operations directly read from the store after applying all committed entries.`

6 Execution Workflow

6.1 Node Startup

```
$ go run node/main.go 2 127.0.0.1:8000 127.0.0.1:8001 127.0.0.1:8002
$ go run node/main.go 1 127.0.0.1:8000 127.0.0.1:8001 127.0.0.1:8002
$ go run node/main.go 0 127.0.0.1:8000 127.0.0.1:8001 127.0.0.1:8002
```

6.2 Client Execution

```
$ go run client/main.go 127.0.0.1:8000 127.0.0.1:8001 127.0.0.1:8002
    Leader is: 127.0.0.1:8000
    Command {Put foo bar} committed at index 0
    Command {Append foo 123} committed at index 1
       Get "foo" => "bar123"
```

7 Testing and Observations

- Startup delays ensure elections occur after all nodes are online.
- Heartbeats from the leader prevent unnecessary elections.
- Commands replicate and commit successfully across all nodes.
- Client consistently finds and interacts with the leader.

8 Challenges and Fixes

- Race on Startup: Election timeouts were too short. Fixed by increasing to 5--6s.
- Multiple main() functions: Resolved by using separate folders for client and nodes.
- Incorrect Get Result: KVStore wasn't updated before Get. Fixed by calling ApplyCommitted before reading.

9 Future Work

- Add persistent log storage to survive crashes.
- Handle leader failure and re-election robustly.
- Provide REST APIs or Web UI for user-friendly interaction.
- Visualize logs and leader status in real-time.

10 Conclusion

This project demonstrates a practical understanding of distributed consensus and fault-tolerant design. Through the Raft protocol, we successfully built a mini distributed key-value store with election, replication, and command handling capabilities. Go's concurrency and RPC made it an ideal language for this implementation.