Introduction to Go - Final Project

Distributed Key-Value Store with Raft Consensus

Introduction

Welcome to your final project! This assignment is designed to push your understanding of Go to its limits by building a **Distributed Key-Value Store**. Distributed systems are where Go truly shines due to its first-class support for concurrency, networking, and efficient binary serialization.

In this project, you will move beyond a single-server application and build a cluster of nodes that must agree on data state even if some nodes fail. You will implement a simplified version of a consensus algorithm (like Raft) to ensure data consistency across multiple instances.

This project will challenge you to:

- Implement complex **Network Communication** between peer nodes.
- Master **Channels and Select** statements for coordination.
- Use **Custom Serialization** (like GOB or JSON) for log replication.
- Handle **Partial Failures** and network partitions.
- Implement a **Distributed Background Job** (log compaction).

Project Overview

You will create a distributed system where multiple Go processes work together to store data. Each node will provide a REST API for users to interact with, but behind the scenes, the nodes will communicate with each other to replicate data.

The system will consist of:

1. **The Storage Engine:** A thread-safe in-memory map.
2. **The Consensus Module:** Logic that handles leader election and log replication.
3. **The Peer Manager:** Maintains a list of active cluster members.
4. **Log Compaction:** A periodic background task that cleans up old command history to save memory.

Project Objectives

By completing this project, you will:

- Use **TCP/UDP/RPC** for inter-node communication.
- Implement **State Machines** to manage node roles (Follower, Candidate, Leader).

- Apply **Advanced Concurrency** patterns to manage heartbeat timers and request-vote cycles.
- Use context.WithTimeout to manage network latency and node unresponsiveness.
- Implement **Persistent Logs** to allow nodes to recover their state after a crash.

Project Requirements

1. Design Data Models

Define the structures for the cluster state and the replication logs.

Node Struct

```go
type Node struct {
    ID      int        `json:"id"`
    Address string     `json:"address"` // e.g., "localhost:8080"
    Peers   []string   `json:"peers"`
    Role    string     `json:"role"`    // Leader, Follower, Candidate
    Log     []LogEntry `json:"log"`
    CommitIdx int      `json:"commit_index"`
    mu      sync.RWMutex
}
```

LogEntry Struct

```go
type LogEntry struct {
    Term    int         `json:"term"`
    Command string      `json:"command"` // e.g., "SET key value"
    Key     string      `json:"key"`
    Value   interface{} `json:"value"`
}
```

2. Define Interfaces for Replication

Create interfaces to abstract the consensus mechanism.

Consensus Interface

```go
type Consensus interface {
    RequestVote(term int, candidateID int) (granted bool)
```

```
    AppendEntries(term int, leaderID int, entries []LogEntry) error
    StartElection()
}
```

3. Implement RPC Communication

Instead of standard HTTP for everything, use Go's net/rpc package or raw net connections for high-speed node-to-node communication.

- Nodes must "heartbeat" their peers to maintain leadership.
- If a heartbeat is missed, a new election must be triggered concurrently.

4. The Client API

Expose a standard RESTful API for the end-user:

- GET /get/{key}: Retrieve a value (can be served by any node or forwarded to the leader).
- POST /set: Submit a new key-value pair (must be handled by the leader and replicated).
- GET /cluster/status: View the current leader and health of all nodes.

5. Periodic Log Compaction

To prevent the log from growing infinitely, implement a background goroutine:

- Runs every X minutes.
- Identifies entries that have been committed and applied.
- Snapshots the current state and clears the log.

Grading Criteria

Criteria	Points
Replication Logic: Data is successfully copied to at least 2/3 of the nodes.	35
Leader Election: System correctly elects a new leader if the current one dies.	25
Concurrency & Safety: No race conditions during concurrent "SET" operations.	15
Networking: Robust handling of connection timeouts and peer retries.	10

Persistence: Nodes can reload their state from a local JSON/Binary file.	10
Documentation: Clear explanation of how to spin up a 3-node cluster.	5
Total	**100**

Example Scenario

1. **Node A (Leader)** receives SET name "Gemini".
2. **Node A** adds the entry to its local log (uncommitted).
3. **Node A** concurrently sends AppendEntries RPCs to **Node B** and **Node C**.
4. Once **Node B** responds with success, **Node A** "commits" the entry and responds to the user.
5. **Node C** (which was offline) comes back online and synchronizes its log automatically with the leader.

Optional Enhancements

- **Partition Tolerance:** Handle the "Split Brain" scenario where two nodes think they are the leader.
- **Custom Binary Protocol:** Use encoding/gob instead of JSON for faster replication.
- **Dynamic Membership:** Implement an endpoint to add a 4th or 5th node to a running cluster.

Good luck, and happy coding!