Contents

Notes

This document contains compiled notes from various modules.

Generated automatically from individual self note files.

Chapter 1: Distributed Key Storage Systems

Date: 22-09-2025 Instructor: Sounish Nath

Main points

```
* Basic amino acid structure
* Overview of amino acid
* Peptide, bonds and polypeptide confirmations restrictions
* Primary, secondary, tertiary structure of proteins in body
* Diseases caused of manifold of proteins
```

Basics of Amino Acid

```
* Peptide, bonds and polypeptide confirmations restrictions
* L and D isomers, L are used for proteins
```

Chapter 2: Raft Consensus Distributed Key store

Date: 22-09-2025 Author: Sounish Nath

This project is an implementation of a sharded and replicated in-memory key-value store in Golang. It uses the Raft consensus algorithm for fault tolerance and gRPC for communication between nodes.

Features

- **Distributed:** Keys are distributed across multiple nodes.
- **Fault-Tolerant:** The system can tolerate node failures as long as a quorum of nodes is alive.
- **Strongly Consistent:** All writes are strongly consistent, thanks to the Raft consensus algorithm.
- **Eventual Consistency:** The system uses a gossip protocol to ensure that all nodes eventually converge to the same state.
- **Cluster Management:** Nodes can dynamically join the cluster.

Architecture

The project is structured into the following components:

- `main.go`: The main application file, responsible for starting the gRPC server and the Raft instance.
- `raft.go`: Implements the Raft Finite State Machine (FSM), which applies commands from the Raft log to the in-memory store.
- `ring.go`: A from-scratch implementation of a consistent hashing ring to map keys to nodes.
- `gossip.go`: Contains the implementation of the gossip protocol for state reconciliation.
- `proto/kv.proto`: The Protocol Buffers definition for the gRPC service, defining the `Put`, `Get`, `Join`, and `Gossip` RPCs.
- `cli/main.go`: A simple command-line client for interacting with the cluster.

Gossip Protocol

The system uses a gossip protocol to ensure eventual consistency across all nodes. Each node periodically selects a random peer and shares its current state. When a node receives a gossip message, it merges the received state with its own using a "last-write-wins" strategy. This mechanism allows information to propagate through the cluster, ensuring that all nodes eventually converge to a consistent view of the data.

Thread-Safe and Concurrent Design

1. **Core Key-Value Store:** The central data store (`Store` struct in `main.go`) is designed to be thread-safe. It uses a `sync.RWMutex`, which is the correct synchronization primitive for a data structure that has both reads and writes. This ensures that:
   - Multiple goroutines can safely read data concurrently (`Get` method).
   - Only one goroutine can write data at a time, preventing data corruption (`Put` method).
2. **Concurrent Goroutines for High Load:** The application effectively uses goroutines to handle concurrent operations:
   - **gRPC Server:** The main gRPC server runs in its own goroutine. The gRPC framework itself is designed for high performance and handles each incoming client request in a separate goroutine, allowing the server to manage many simultaneous connections.
   - **Gossip Protocol:** The gossiping mechanism runs in a dedicated background goroutine (`startGossip` in `main.go`), periodically communicating with other nodes without blocking the main application flow.
   - **Raft Consensus:** The project uses the `hashicorp/raft` library, a mature and battle-tested implementation of the Raft consensus algorithm. This library is inherently concurrent and designed to manage a distributed state machine in a fault-tolerant way.

How to Run

Prerequisites

- Go

- Protocol Buffers Compiler (`protoc`)

1. Build the Project

First, ensure all dependencies are downloaded:

```
go mod tidy
```

Generate the GRPC protobufs

```
export PATH="$PATH:$(go env GOPATH)/bin" && protoc --go_out=. \
--go_opt=paths=source_relative --go-grpc_out=. \
--go-grpc_opt=paths=source_relative proto/kv.proto
```

2. Start the Cluster

You will need to open three separate terminals to run a 3-node cluster.

Terminal 1: Start the first node (bootstrap node)

```
echo ("single node")
go run . --bootstrap
```

OR

```
go run . --port=50051 --raft_port=12001 --node_id=node1 \
--raft_dir=/tmp/raft1 --bootstrap=true
```

Terminal 2: Start the second node and join the cluster

```
go run . --port=50052 --raft_port=12002 --node_id=node2 \
--raft_dir=/tmp/raft2 --join_addr=localhost:50051
```

Terminal 3: Start the third node and join the cluster

```
go run . --port=50053 --raft_port=12003 --node_id=node3 \
--raft_dir=/tmp/raft3 --join_addr=localhost:50051
```

3. Interact with the Cluster

Open a fourth terminal to run the client application.

```
go run cli/main.go
```

This will send a `Put` request to store a key-value pair and then a `Get` request to retrieve it.

4. Test Fault Tolerance

To test the system's fault tolerance, you can stop one of the nodes (e.g., by pressing `Ctrl+C` in its terminal). Then, run the client again. The `Put` and `Get` operations should still succeed as long as a majority of the nodes (a quorum) are still running.

```
go test -v
```

Performance

Benchmarks were run to measure the performance of `Put` and `Get` operations. You can run the benchmarks yourself using the following command:

```
go test -bench=. -benchmem -run=^$
```

Results

Here are the results from a sample run:

```
BenchmarkPut-8        20599         57753 ns/op         8949 B/op         171 allocs/op
BenchmarkGet-8        21543         59570 ns/op         8949 B/op         171 allocs/op
```

Understanding the Metrics

- **`ns/op` (Nanoseconds per operation):** This is the average time it took to execute a single operation. A lower number indicates better performance.
- **`B/op` (Bytes per operation):** This represents the average amount of memory allocated per operation. A lower number is better, as it indicates more efficient memory usage.
- **`allocs/op` (Allocations per operation):** This is the average number of memory allocations made per operation. Fewer allocations generally lead to less work for the garbage collector and better performance.

Deriving Latency and Throughput

From these metrics, we can derive the latency and throughput of the system.

- **Latency:** The `ns/op` value directly represents the average latency for a single operation.
- **Throughput:** Throughput is the number of operations the system can handle per second. It can be calculated as the inverse of the latency (`1,000,000,000 ns/second / ns/op`).

System Performance

Based on the benchmark results, the current system performance is as follows:

- **Put Operation:**
  - **Latency: 57,753 ns/op** (approximately 57.75 microseconds)
  - **Throughput: ~17,315 operations/second**
- **Get Operation:**
  - **Latency: 56,783 ns/op** (approximately 56.78 microseconds)
  - **Throughput: ~17,611 operations/second**

Chapter 3: Closing Notes

> Date: 22-09-2025 Author: Sounish Nath

Based on my analysis of the codebase, here is the answer to your question:

Yes, the project is largely implemented to be thread-safe and uses concurrent goroutines to handle load, but there is a potential race condition in one part of the code.

```
echo "Hope you loved it. Feel free to create your valuable Notes
```

Sounish Nath