Distributed Database System Report

May 7, 2025

1 Introduction

This project implements a distributed database system using Go, as specified in the project requirements. The system features a master-slave architecture with MySQL as the backend database, supporting dynamic database and table creation, CRUD operations, and data replication across nodes. The master node coordinates critical operations, while slave nodes replicate data for redundancy. A client GUI, built with the Fyne framework, provides a user-friendly interface for database interactions. This report details the architecture, design choices, and challenges encountered during development, directly referencing the provided master and slave code.

2 Architecture Overview

The system is structured around a master-slave model, with the master node (`main.go`) managing a MySQL database and coordinating operations, and the slave nodes replicating data. The architecture, implemented in Go, consists of three nodes (minimum requirement) communicating via HTTP. The master node handles database creation (`CreateDatabase`), table creation (`CreateTable`), and replication (`ReplicateToPeers`), while both master and slaves support queries like insert (`InsertRecord`), select (`SelectRecords`), update (`UpdateRecord`), and delete (`DeleteRecord`). The slave code (`main.go`) includes a Fyne-based GUI (`GUIInterface`) for client interactions.

Key components include:

- Database Module: The `Database` struct manages MySQL connections, enabling dynamic database switching and operations (e.g., `CreateDatabase`, `InsertRecord`).
- Node Module: The `Node` struct defines master and slave behavior, with the master replicating operations to peers (`ReplicateToPeers`) and monitoring peer health (`MonitorPeers`).
- Server Module: The `Server` struct exposes HTTP endpoints (e.g., `/create-db`, `/select`) for node communication, as seen in `Start`.
- Client Module: The slave code's `Client` struct and `GUIInterface` provide a GUI for users to interact with the master node, supporting dynamic table and record management.

3 Design Choices

The following design choices were made to implement the system's functionality:

- Master-Slave Architecture: The `Node` struct's `IsMaster` field restricts database creation and dropping to the master (`CreateDatabase`, `DropDatabase`), ensuring centralized control. All nodes handle queries, as seen in `InsertRecord` and `SelectRecords`.
- MySQL Backend: The `Database` struct uses the `go-sql-driver/mysql` package, connecting without a specific database (`NewDatabase`) to support dynamic creation (`CreateDatabase`).
- HTTP Communication: HTTP POST requests (`ReplicateToPeers`) and endpoints (`createDBHandler`, `insertHandler`) enable node communication, with JSON payloads for operations (e.g., `json.Marshal`).
- Concurrency Control: The `Node` struct's `mu` mutex (`sync.Mutex`) ensures thread safety during operations, as seen in `CreateTable` and `InsertRecord`.
- Fyne GUI: The slave code's `GUIInterface` uses Fyne for a cross-platform GUI, with dynamic form generation (e.g., `insertBtn`) for flexible record insertion and updates.
- Replication: The master's `ReplicateToPeers` function sends operations to slaves via HTTP, ensuring data consistency across nodes, as required.
- Error Handling: Comprehensive error handling (e.g., `http.Error` in handlers, retries in `GetDatabases`) and logging (`log.Printf`) enhance robustness.

4 Challenges and Solutions

Development presented several challenges, addressed as follows:

- Dynamic Database Management: MySQL connections typically bind to a database, complicating dynamic operations. The `NewDatabase` function connects without a database, and `USE` statements (`Exec("USE ...")`) enable switching, as seen in `CreateTable` and `InsertRecord`.
- Replication Consistency: Ensuring consistent replication across nodes was challenging. The `ReplicateToPeers` function uses a `sync.WaitGroup` and mutex (`mu.Lock()`) to synchronize operations, though it lacks strong consistency guarantees (e.g., two-phase commit).
- Network Reliability: HTTP requests could fail due to network issues. The `client` in `Node` has a 5-second timeout (`http.ClientTimeout: 5 * time.Second`), and errors are logged without halting operations (`errChan`).
- GUI Responsiveness: Fetching databases (`GetDatabases`) and tables (`GetTables`) could delay the GUI. The slave code runs these in a goroutine (`go func() ...`) to update the UI asynchronously.
- Fault Tolerance: The optional fault tolerance feature is partially implemented via `MonitorPeers`, which checks peer health every 10 seconds. Full master failover

(e.g., leader election) was not implemented due to complexity.

- Security: The system lacks authentication and encryption, as seen in open HTTP endpoints (e.g., `/insert`). This was acceptable for the project but requires TLS and authentication for production use.

5 Conclusion

The distributed database system, implemented in Go, fulfills the project requirements by providing a master-slave architecture with MySQL integration, HTTP-based replication, and a Fyne GUI. The master node's `CreateDatabase` and `ReplicateToPeers` functions, combined with the slave's `GUIInterface`, enable dynamic database management and user-friendly interactions. Challenges like dynamic database switching and replication consistency were addressed, though fault tolerance and security remain areas for improvement. Future enhancements could include a consensus algorithm (e.g., Raft), authentication, and master failover to enhance reliability and security.