Project Title: Distributed Train Ticket Booking System

Team Number: 7

Team Members:

1. **Ashish Shetty** (2025H1030064P)
2. **S Rishab** (2025H1030066P)

Project Overview

This project implements a fault-tolerant, distributed train ticket booking system using gRPC and Raft Consensus Algorithm. The system ensures consistent state replication across multiple nodes, supports client interactions via a Flask web application, and integrates a local LLM server to provide intelligent assistance.

The system consists of three Raft nodes (one leader, two followers), a web-based client interface, a distributed booking service, and an LLM inference backend.

Project Responsibilities

1. Ashish Shetty

- Raft Consensus Implementation
  - Designed and implemented the core Raft algorithm, including leader election, log replication (using AppendEntries), and state persistence to disk.
  - Built replication pipelines using AppendEntries, nextIndex, and matchIndex tracking.
  - Implemented fault-tolerance features such as leader failure detection (via heartbeats) and log reconciliation on node restart.
- Project Structure & Architecture
  - Designed the overall system architecture diagram and interaction workflow.
  - Organized the server-side codebase into coherent, decoupled modules for Raft operations, the application service, database models, and utilities.
- Database Layer
  - Implemented robust SQLite-based data models and transactional safety.
  - Ensured all write operations use Raft logs and remain consistent across nodes.
  - Implemented the Write Ahead Logs to ensure atomicity and durability by recording changes before they are applied to the database.
- LLM Integration
  - Implemented gRPC + HTTP pipeline between Raft leader and a locally running LLM server.
  - Ensured that all LLM queries are routed through the leader, maintaining consistency and single-writer logic.
  - Added fallback and timeout-handling mechanisms to maintain system responsiveness.

2. S Rishab

- Client UI Development
    - Built a user-friendly web-based interface using Flask.
    - Ensured correct validation, error handling, and smooth user flow.
    - Ensure correct client-side routing and calling of gRPC client functions through it.
- gRPC Client–Server Interaction
    - Developed the entire client-side gRPC logic for communication with Raft nodes.
    - Implemented:
        - Automatic leader redirection
        - Retry loops when nodes are down
        - Error formatting and clean user feedback
        - Failover handling when the leader crashes
    - Ensured the redirection of the request from the client to the leader node for each request.
- Database & Booking Service Integration
    - Built the booking service that interacts with Raft that provides the functionality of Login, Register, Booking Tickets, Adding Trains and Services.
    - Added safeguards against Lost Updates, Duplicate Transactions and Database Locking issues.
    - Enforced Determinism and ensured that all nodes apply the exact same IDs and data for every log entry.
- gRPC Service & API Contract (.proto)
    - Designed and implemented the strict, language-agnostic API contract (.proto file) that served as the "single source of truth" for all communication within the distributed system.
    - Designed the AskBot RPC as a server-streaming call to enable a responsive, token-by-token typewriter effect in the UI.

Key Challenges

- Preventing Split-Brain Problem (Two Leaders)
    - Upon leader failure, two follower nodes would simultaneously time out and elect themselves as leaders for new, competing terms.
    - The RequestVote function in raft_node.py was refactored to include the log safety check, comparing both last_log_term and last_log_index. This ensures a node with a stale log can never win an election.
- To ensure replication across the nodes
    - The BookingService was violating the "Log-First" principle. It was performing a double-write, that is, first writing to its own database, and then sending the command to the Raft log.
- Enforcing Determinism
    - Critical data, such as booking_id, service_id, and payment_id, was inconsistent across the cluster. A booking made on the leader would be replicated on followers with different IDs, corrupting the database and making future operations like ProcessPayment impossible.
    - To rectify it, the system was refactored to enforce strict determinism.