Project Reflection: Distributed Ticket Booking System

Course: Advanced Operating Systems

Project Team: Kalp Dalsania (2025H1030209P)

Saksham Singhal (2025H1030208P)

Date: November 15, 2025

---

1. Project Overview and Goals

The project focused on designing and implementing a **distributed ticket booking platform** to meet critical requirements for **strong consistency** and **high availability**. The fundamental goal was to leverage the **Raft consensus algorithm** to eliminate concurrency faults, specifically preventing **double-bookings**, across a three-node booking cluster.

---

2. System Architecture and Component Ownership

The architecture is built on **gRPC-based microservices**, ensuring modularity and performance.

Team Member	ID	Core Responsibilities
Kalp Dalsania	**2025H1030209P**	**Raft Consensus Engine, Log Replication, Leader Election, State Persistence, Consensus Testing**
Saksham Singhal	**2025H1030208P**	**Booking Logic Orchestration, Authentication, Payments, Chatbot Services**

Key Technical Contribution

- **Raft Consensus Layer & Testing (Kalp Dalsania):** Implemented the core Raft protocol (raft.py), managing log entries, and applying committed commands to the deterministic State Machine (state_machine.py). This layer guarantees that all nodes commit state changes in the exact same order.

    - **Testing Suite:** Developed the critical testing utilities used to validate the consensus engine's functionality, including **concurrency safety** (stress_test.py) and **fault tolerance/leader failover** (test_raft_failover.py, test_leader_election.py).
    - **Timing Detail:** The implementation uses an explicit **Heartbeat Interval of 50ms** and a randomized **Election Timeout between 150-300ms** to minimize split votes and ensure rapid fault detection.
- **Application & Orchestration (Saksham Singhal):** Developed all external-facing **microservices (Authentication, Payment, Chatbot).** This included the **Booking Service logic** (booking_service.py) responsible for coordinating the **multi-step transaction**, **Session Validation, Payment Processing Raft Command Proposal**.

---

3. Technical Validation

System robustness and core distributed properties were validated through targeted Python test scripts.

3.1. Concurrency Safety (Stress Testing)

Test Metric	Validation Logic	Result
Zero Double-Bookings	`if len(successes) == 1 and len(failures) == len(results) - 1:`	Consistently validated as **1 Successful Booking** and **29 Failures**, confirming the serialization of requests mandated by Raft.

3.2. Fault Tolerance (Leader Failover)

Test Objective	Validation Steps	Result
HA Confirmation	1. Book Seat 1 on initial Leader. 2. `kill_node(leader_id)` terminates process. 3. A new Leader is confirmed (`new_id != leader_id`). 4. Book Seat 2 on the new Leader, verifying read/write access is restored.	**Successful election** and subsequent successful write operation, demonstrating rapid recovery and state continuity.

---

4. Challenges and Professional Learning

- **Raft Implementation Complexity:** The necessity of debugging asynchronous Raft logic and meticulous log conflict resolution reinforced the importance of robust **invariant testing** and precise state machine logic, particularly in handling the AppendEntries logic to ensure log consistency across all nodes.
- **Microservice Orchestration:** Managing asynchronous gRPC calls within the sequential booking flow (AuthService -> PaymentService ->RaftNode.propose) required specialized error handling to prevent state corruption (e.g., ensuring a payment failure halts the transaction prior to a Raft log proposal).
- **Design Trade-off (Chatbot):** The decision to use a **template-based intent-classification model** over an exploratory generative AI was a strategic choice to prioritize **determinism** and **high functional accuracy** for predictable booking queries, aligning with enterprise-level service reliability requirements.
