PBLE 2

Semester	B.E. Semester VIII – Computer Engineering
Subject	Distributed Computing Lab
Subject Professor In-charge	Dr. Umesh Kulkarni
Assisting Professor	Prof. Prakash Parmar
Academic Year	2024-25

Student Name	Deep Salunkhe
Roll Number	21102A0014

Title: Solving Synchronization Issues in a Distributed Ticket Booking System.

Problem Statement

A team is developing a distributed ticket booking system for a global event. The system is hosted across multiple servers in different regions to handle high user traffic. Each server maintains a replica of the ticket inventory to reduce latency and improve availability. However, three major synchronization issues arise:

1. **Overbooking:** Multiple users are allocated the same ticket due to race conditions between servers.

2. **Inconsistent State:** Some servers show available tickets while others show sold-out status, causing confusion among users.

3. **Delayed Updates:** Network delays cause slow propagation of ticket availability updates, leading to inaccurate inventory status.

Proposed Solutions

To address these synchronization issues, we propose a combination of **distributed database techniques, consensus algorithms, and event-driven architectures**.

---

1. Solving Overbooking

1.1 Distributed Locking Mechanism

- Implement **distributed locks** using a system like **Redis (RedLock algorithm)** or **Zookeeper**.
- When a user requests a ticket, a lock is placed on the seat for a short duration to prevent other servers from allocating the same ticket.
- If the user completes the transaction, the ticket is confirmed; otherwise, the lock expires.

1.2 Optimistic Concurrency Control (OCC)

- Use **Optimistic Concurrency Control** with **Versioning**.
- Every booking attempt checks if the ticket's version matches the current version in the database.
- If a mismatch occurs (i.e., another server has already booked it), the transaction fails, prompting the user to retry.

1.3 Eventual Consistency with Strong Read Guarantees

- Use **distributed transactions** (e.g., **Two-Phase Commit (2PC)** or **SAGA pattern**) to ensure that a ticket allocation request is confirmed across all servers before committing.
- A ticket is considered sold only after consensus is reached across servers.

---

2. Solving Inconsistent State

2.1 Use of Distributed Databases

- Implement **global consensus** using **Paxos or Raft** in a distributed database like **CockroachDB, Spanner, or DynamoDB**.
- This ensures that all replicas see a consistent ticket count.

2.2 Read-Your-Own-Writes Consistency

- Ensure users always see their most recent transaction by implementing **session consistency**.

- Each user request can be directed to the last server that processed their request to avoid discrepancies.

2.3 Conflict Resolution Strategies

- Use **CRDTs (Conflict-free Replicated Data Types)** or **event sourcing** to handle conflicting states.
- If two servers mark the same ticket as available at the same time, conflict resolution policies (e.g., last-write-wins or majority consensus) ensure consistency.

---

3. Solving Delayed Updates

3.1 Real-time Event Propagation with Pub/Sub

- Use **event-driven architecture** with **Kafka, RabbitMQ, or AWS SNS/SQS** to broadcast ticket availability changes instantly to all servers.
- Each server subscribes to ticket updates, ensuring near real-time synchronization.

3.2 Database Change Streams

- Leverage **Change Data Capture (CDC)** using **Debezium** or **DynamoDB Streams** to listen for updates and sync changes across all replicas.
- Ensures that as soon as a ticket is booked, all other servers receive the update.

3.3 Vector Clocks for Causal Ordering

- Use **vector clocks** to track event ordering and prevent outdated updates from overriding newer ones.

Implementation Strategy

1. System Architecture

- **Backend Services:** Microservices-based architecture with dedicated services for booking, payment, and notifications.
- **Database Layer:** Uses a distributed database with ACID-compliant transactions

for critical operations.

- **Messaging Layer:** Uses event-driven updates for state synchronization.
- **Cache Layer:** Uses **Redis or Memcached** to reduce read latency while ensuring cache invalidation upon updates.

2. API Design

- **Book Ticket API:**
  - Implements distributed locking and OCC.
  - Uses event-based consistency updates.
- **Check Availability API:**
  - Uses a read-through cache with strong consistency.
- **Confirm Booking API:**
  - Uses a distributed transaction mechanism.

3. Performance Optimization

- **Rate limiting & throttling** to prevent excessive API calls.
- **Load balancing with sticky sessions** to improve user experience.
- **Edge caching** to reduce redundant calls to the main database.

Conclusion

By integrating **distributed locking, consensus mechanisms, event-driven updates, and strong consistency models**, we can effectively mitigate overbooking, inconsistent states, and delayed updates. The proposed system ensures a seamless and reliable ticket booking experience for a global audience while maintaining high availability and scalability.

Technology Stack Recommendation

Component	Suggested Tools/Technologies
Database	CockroachDB, DynamoDB, Spanner

Distributed Locking	Redis (RedLock), Zookeeper
Event Propagation	Kafka, RabbitMQ, AWS SNS/SQS
Load Balancing	Nginx, AWS ALB, Cloudflare
Concurrency Control	Optimistic Locking, 2PC, SAGA
Conflict Resolution	CRDTs, Event Sourcing

This approach ensures high availability, fault tolerance, and a smooth booking experience for all users.