Distributed RideShare System – Project Report

Group Number: 25

Team Members: Rohan Dave and Shreyas Prakash

Overview

The Distributed RideShare System is a concurrent, multi-service simulation inspired by Uber. It demonstrates distributed-systems fundamentals—multi-process architecture, inter-service communication, and concurrency using multi-threaded servers. Each service (Booking, Driver, Pricing, Map, Payments) runs independently and exchanges messages over TCP sockets. The project focuses on thread-safe concurrency and performance under multiple simultaneous client requests like in programming assignment 2.

Goals

- Implement independent service processes communicating via sockets.
- Introduce multi-threaded request handling in key services (Booking, Driver).
- Ensure thread-safe access to shared in-memory data.
- Evaluate throughput and latency under concurrent requests.
- Demonstrate fault containment—crashes in one service do not stop others.

System Assumptions

- **Failures:** Each service may fail independently. Retry mechanisms and timeouts are implemented at the Booking Service level. If a dependent service (e.g., Pricing or Payments) fails, the request is retried up to three times before being marked as a failure. This ensures basic fault tolerance and resiliency under transient network or service outages.
- **Workload:** Simulated load is generated by 100–1000 concurrent clients (riders) using the Client Simulator. Drivers continuously update their status with the Driver Service, generating approximately 10–20 updates per second. This workload models a real-world concurrent environment with multiple users and live driver updates.
- **Users:** We assume 1,000 active riders and 200 drivers in the simulation. Each rider can request multiple rides during the test run, allowing scalability and concurrency testing under realistic load conditions.

System Design

The system follows a microservices architecture composed of multiple independent components, each responsible for a specific function within the ride-sharing workflow. The services are as follows:

- **Booking Service:** Acts as the central coordinator for ride requests. It receives client booking requests, communicates with other services to find available drivers, calculates fares, and finalizes ride details.
- **Driver Service:** Manages driver information, including availability and location data. It handles concurrent updates from multiple clients and driver simulators, ensuring thread-safe access to shared data structures.
- **Pricing Service:** Calculates estimated fares based on distance and duration information obtained from the Map Service. It supports concurrent pricing queries from the Booking Service.
- **Map Service:** Computes distances and estimated travel times between pickup and drop-off locations. It can interface with external APIs (e.g., Google Maps API) or use simulated distance calculations for testing.

- **Payments Service:** Simulates fare payments and transaction logging. It ensures atomic processing of payment requests even under concurrent booking scenarios.

- **Driver Simulator:** Continuously updates driver positions in the Driver Service to mimic real-time driver movement and availability.

- **Client Simulator:** Generates concurrent rider booking requests to test system scalability, latency, and throughput under load.

Evaluation Plan

Evaluation will be conducted on the Khoury Linux cluster by deploying each service as an independent process on separate nodes. The evaluation focuses on assessing the system's concurrency, responsiveness, and fault tolerance under different workload intensities.

Experimental Setup

- Each microservice (Booking, Driver, Pricing, Map, and Payments) will run on a separate node or port.
- The Client Simulator will generate concurrent booking requests to vary workload intensity.
- System logs will capture request timestamps and inter-service message delays for performance measurement.

Metrics

- **End-to-End Latency:**
  Measure total response time from the client sending a booking request to receiving the final confirmation. (Client → Booking → Driver/Pricing/Payments → Client).

- **Throughput Across Concurrency Levels:**
  Measure the number of completed ride requests per second as the number of concurrent client threads increases.

- **System Scalability:**
  Evaluate performance improvement when multiple Booking and Map Service replicas are deployed to handle higher request rates.

- **Fault Tolerance:**
  Simulate service crashes or restarts (e.g., killing one microservice process) and observe the system's ability to continue functioning with minimal interruption.

- **Cache Effectiveness (Map Service):**
  Compare response times and system throughput with and without caching enabled in the Map Service to evaluate the impact of caching on latency and performance.

Evaluation Procedure

- Deploy each service process on a separate Khoury cluster node or unique port.
- Use the Client Simulator to send booking requests at varying concurrency levels (e.g., 1, 5, 10, 20, 50 clients).
- Record response times, throughput, and system logs for each configuration.
- Repeat the experiment under simulated fault conditions (service restarts) and with/without caching enabled.
- Analyze results and compare system behavior under different loads and fault scenarios.

Step-by-Step Plan & Timeline (Subject to change)

- **Week 1**: Implement all core services (Booking, Driver, Pricing, Map, Payments) with socket communication and verify a basic end-to-end booking flow.
- **Week 2**: Add multi-threading, thread-safe shared data handling, and integrate client/driver simulators for concurrent request processing.
- **Week 3**: Deploy services on the Khoury Linux cluster, run performance and fault-tolerance evaluations, and finalize results for the report

Conclusion

This project models a distributed, Uber-like architecture that incorporates real-world design principles such as service decomposition, fault isolation and concurrency handling. Retry mechanisms ensure resiliency, and the use of simulators for both clients and drivers provides a realistic test environment. The system enables a practical evaluation of distributed concepts like scalability, eventual consistency, and latency optimization under concurrent workload.