Homework_4

Team_11

Q3 MyUber (35 Points)

Detailed Report

Vilal Ali, 2024701027, (Questions 2 and 3)

Kapil Rajesh Kavitha - 2021101028 (Questions 1 and 4)

1. Introduction

The MyUber Ride-Sharing System is designed to be a secure, distributed system that enables riders to request rides and drivers to accept, reject, or complete these ride requests. The system leverages gRPC for high-performance communication and implements SSL/TLS with mutual authentication (mTLS) to ensure secure communication between riders, drivers, and the server. The system handles ride assignments, timeout and rejection scenarios, driver availability, and client-side load balancing across multiple servers for scalability and fault tolerance.
This report details the implementation of the system, including API design, security features, interceptors, load balancing mechanisms, and timeout handling.

2. Architecture Design

2.1 System Overview

The MyUber system consists of:

- **Riders:** Clients who request rides by providing their pickup and destination locations.
- **Drivers:** Clients who accept or reject ride requests based on availability and status.
- **gRPC Servers:** Multiple distributed servers that handle ride requests, load balance across servers, and communicate securely with riders and drivers using SSL/TLS.
- **SSL/TLS:** Provides encryption and authentication, ensuring that only valid riders and drivers can participate.

2.2 Communication Flow

1. **Rider requests a ride:** The rider sends a ride request with their pickup and destination location.
2. **Driver accepts/rejects the ride:** The server assigns the ride to an available driver. The driver either accepts or rejects the request within a specified timeout.
3. **Ride reassignment:** If the driver does not respond or explicitly rejects the request, the system automatically reassigns the ride to another available driver.
4. **Ride completion:** Once the driver completes the ride, the ride status is updated, and the driver becomes available for new requests.

2.3 Server Architecture

The system uses a multi-server setup with gRPC clients connecting to multiple backend servers. Requests are load balanced across available servers using a round-robin algorithm, ensuring even distribution of ride requests.

3. API Design

The API methods are defined using gRPC .proto files. The key services and messages are as follows:

3.1 Rider Services

- **RequestRide(RideRequest) → RideResponse:** Riders submit a ride request by providing their pickup location and destination.

  RideRequest:

```
message RideRequest {
        string pickup_location = 1;
        string destination = 2;
}
```

RideResponse:

```
message RideResponse {
        string status = 1;
        string driver_id = 2;
}
```

- **GetRideStatus(RideStatusRequest) → RideStatusResponse:** Riders can check the status of their requested or ongoing rides.

```
message RideStatusRequest {
        string ride_id = 1;
}

message RideStatusResponse {
        string status = 1; // e.g., ASSIGNED, IN_PROGRESS,
        COMPLETED, CANCELED
}
```

3.2 Driver Services

- **AcceptRide(AcceptRideRequest) → AcceptRideResponse:** Drivers can accept a ride request assigned to them. The system waits for a response within a timeout period.

AcceptRideRequest and AcceptRideResponse:

```
message AcceptRideRequest {
        string ride_id = 1;
        string driver_id = 2;
}
message AcceptRideResponse {
        string status = 1; // e.g., ACCEPTED, TIMEOUT
}
```

- **RejectRide(RejectRideRequest) → RejectRideResponse:** Drivers can explicitly reject a ride if they are unavailable or unwilling to accept it. The ride is automatically reassigned.
  RejectRideRequest and RejectRideResponse:

```
message RejectRideRequest {
        string ride_id = 1;
        string driver_id = 2;
}

        message RejectRideResponse {
        string status = 1; // e.g., REJECTED
}
```

- **CompleteRide(RideCompletionRequest) → RideCompletionResponse:** Drivers can mark the ride as complete after dropping off the rider.
  RideCompletionRequest and RideCompletionResponse:

```
message RideCompletionRequest {
        string ride_id = 1;
        string driver_id = 2;
}

message RideCompletionResponse {
        string status = 1; // e.g., COMPLETED
}
```

4. SSL/TLS Authentication

4.1 Mutual TLS (mTLS) Setup

To ensure secure communication, mutual TLS is implemented. Both the server and client authenticate each other by verifying their SSL certificates. The system uses the following certificates:

- **ca.crt:** Certificate Authority that signs both server and client certificates.
- **server.crt and server.key:** Used by the server to authenticate with clients.
- **client.crt and client.key:** Used by riders and drivers to authenticate the server.

The gRPC server and client configuration includes the following:

```
# Server-side SSL/TLS configuration
grpc_server = grpc.server(futures.ThreadPoolExecutor())
with open('certs/server.crt', 'rb') as f:
        server_certificate = f.read()
with open('certs/server.key', 'rb') as f:
        private_key = f.read()
with open('certs/ca.crt', 'rb') as f:
        ca_certificate = f.read()

server_credentials = grpc.ssl_server_credentials(
        [(private_key, server_certificate)],
        root_certificates=ca_certificate,
        require_client_auth=True
)
grpc_server.add_secure_port('[::]:50051', server_credentials)
```

4.2 Client Certificate Verification

- Riders and drivers must present valid certificates to be authenticated by the server.
- If a client certificate is invalid or expired, the server rejects the connection.

5. Timeout and Rejection Handling

- A timeout of **10 seconds** is set for drivers to respond to a ride request. If no response is received within this time, the system automatically reassigns the ride to another driver.
- If a driver rejects the ride, the system retries to assign the ride to another available driver. After a maximum number of reassignments, the ride request is canceled.

6. Load Balancing

6.1 Round-Robin Load Balancing

The system uses **round-robin load balancing** to distribute ride requests evenly across multiple gRPC servers. This ensures optimal resource utilization and prevents overloading a single server.

6.2 Custom Load Balancing (Bonus)

For additional flexibility, a custom load balancing algorithm could be implemented that considers factors like server load or geographic proximity.

7. Conclusion

The MyUber Ride-Sharing system successfully implements the core features of a distributed ride-sharing system, including SSL/TLS authentication, timeout handling, and load balancing. The use of gRPC ensures high-performance communication, while interceptors provide role-based authorization and detailed logging. The system is designed for scalability, allowing seamless integration of new servers and clients as the system grows.

Note:

Project Setup and Execution: Instructions for running the project and configuring the environment can be found at **Team_11/Q3/README.md.**

Use Case and Demo Video: The demo video showcasing the use case can be found at **Team_11/Q3/Q3_Demo.mp4**.