# Distributed Systems HW-4 Q3 Report

**Team Number:** 55

**Team Members:** Mitansh Kayathwal, Pradeep Mishra

**Roll Numbers:** 2021101026, 2023801013

# 1. Overview

MyUber is a sophisticated ride-sharing application designed to connect riders with drivers in a seamless, efficient, and secure manner. The application is built on a distributed architecture, leveraging modern technologies and best practices in software design to ensure scalability, reliability, and real-time responsiveness.

## Key Components:

1. Server
2. Driver Client
3. Rider Client

## Core Technologies:

- gRPC (Google Remote Procedure Call)
- Protocol Buffers
- etcd (Distributed Key-Value Store)
- Go (Golang) Programming Language
- TLS (Transport Layer Security)

## Key Features:

- Real-time ride requests and assignments
- Driver availability management
- Ride status tracking
- Secure communication
- Distributed system architecture
- Load balancing and fault tolerance

Now, I will dive deeper into each component and explore the design decisions and implementations.

# 2. Server Implementation

The server is the backbone of my MyUber application, acting as the central coordinator for all ride-sharing activities. It’s key aspects include:

## 2.1 Architecture

The server is implemented in Go, taking advantage of the it's strong concurrency support and efficiency. It uses a multi-threaded approach to handle multiple client connections simultaneously.

### Key Design Decisions:

- **gRPC Server**: The server implements both RiderService and DriverService using gRPC, allowing for efficient, type-safe communication with clients.
- **Stateful Design**: Despite using gRPC, which is typically stateless, the server maintains some state information to manage ongoing rides and driver availability.
- **Distributed Architecture**: The server is designed to work in a distributed environment, with multiple instances able to run concurrently.

## 2.2 Data Management

At the server side, I've used etcd client, a distributed key-value store, for maintaining shared state across multiple server instances. This design choice offers several advantages:

- **Consistency**: Ensures data consistency across all server instances.
- **Fault Tolerance**: Provides a reliable storage mechanism that can survive individual server failures.
- **Real-time Updates**: Allows for real-time propagation of state changes across the system.

### Key Data Structures:

- Ride statuses
- Driver availability
- Ride assignments

## 2.3 Ride Management

The server implements a sophisticated ride management system:

### Ride Request Flow:

1. Receive ride request from a rider
2. Generate a unique ride ID
3. Attempt to assign an available driver
4. If no driver is immediately available, retry with a backoff mechanism
5. Update ride status in etcd
6. Notify the assigned driver
7. Confirm ride assignment to the rider

### Driver Assignment Algorithm:

- The server maintains a list of available drivers
- When a ride request comes in, it iterates through available drivers
- Each driver is given a time window to accept or reject the ride
- If a driver accepts, the ride is assigned; if rejected or timed out, the next driver is tried
- This process continues until a driver is assigned or all available drivers have been tried

### 2.4 Security

Security is a top priority in my MyUber server implementation:

- **TLS**: All communications are encrypted using TLS.
- **Certificate-based Authentication**: Clients (both riders and drivers) are authenticated using client certificates.
- **Role-based Access Control**: The server distinguishes between rider and driver roles, ensuring that each can only access appropriate services.

### 2.5 Scalability and Performance

The server is designed with scalability in mind:

- **Stateless Design**: While some state is maintained in etcd, the server itself is largely stateless, allowing for easy horizontal scaling.
- **Efficient Communication**: gRPC's use of HTTP/2 and Protocol Buffers ensures efficient, low-latency communication.
- **Connection Pooling**: The server maintains a pool of connections to etcd to reduce connection overhead.

### 2.6 Error Handling and Logging

Robust error handling and logging mechanisms are implemented:

- **Graceful Error Handling**: The server is designed to handle various error scenarios gracefully, preventing crashes and ensuring system stability.
- **Comprehensive Logging**: A logging interceptor records all incoming requests and responses, facilitating debugging and system monitoring.
- **Structured Logging**: Log entries are structured for easy parsing and analysis.

## 3. Driver Client Implementation

The driver client is the interface through which drivers interact with the MyUber system. It's designed to be user-friendly while providing real-time updates and efficient communication with the server.

### 3.1 Architecture

The driver client is also implemented in Go, leveraging the same gRPC framework used by the server for consistency and efficiency.

### Key Design Decisions:

- **gRPC Client**: Implements the DriverService client interface generated from the Protocol Buffers definition.
- **Long-lived Connections**: Maintains a persistent connection to all the servers to receive real-time ride requests.
- **Command-line Interface**: Provides a simple, text-based interface for driver interactions.

## 3.2 Core Functionalities

### 3.2.1 Driver Registration

- Upon startup, the client generates a unique driver ID.
- It establishes a secure connection with the server using TLS and client certificate authentication.

### 3.2.2 Ride Request Handling

- The client opens a bi-directional gRPC stream with the server to receive ride requests.
- When a ride request is received, it's presented to the driver with relevant details (pickup location, destination).
- The driver can choose to accept or reject the ride within a specified timeframe.

### 3.2.3 Ride Management

- Once a ride is accepted, the client provides options to:
  - Start the ride
  - Complete the ride
  - Check ride status

### 3.2.4 Driver Availability

- The client allows the driver to set their availability status (available/unavailable).
- This status is communicated to the server to manage ride assignments effectively.

## 3.3 User Interface

The driver client implements a simple command-line interface:

- Menu-driven interaction for ease of use
- Clear prompts for ride acceptance/rejection

- Options to manage ongoing rides
- Ability to view ride history and earnings (if implemented)

### 3.4 Error Handling and Reconnection

The client implements robust error handling:

- Automatic reconnection attempts if the connection to the server is lost
- Clear error messages displayed to the driver
- Graceful handling of server unavailability or network issues

## 4. Rider Client Implementation

The rider client serves as the entry point for users seeking rides. It's designed to be intuitive, responsive, and reliable.

### 4.1 Architecture

Like the driver client, the rider client is implemented in Go and uses gRPC for communication with the server.

### Key Design Decisions:

- **gRPC Client**: Implements the RiderService client interface.
- **Load Balancing**: Incorporates client-side load balancing to distribute requests across multiple server instances.
- **Command-line Interface**: Offers a straightforward, text-based interface for ride requests and status checks.

### 4.2 Core Functionalities

#### 4.2.1 Ride Requests

- Allows users to input pickup and destination locations
- Sends ride requests to the server
- Displays confirmation and ride details upon successful assignment

#### 4.2.2 Ride Status Checking

- Provides an option to check the status of an ongoing ride
- Fetches real-time updates from the server

#### 4.2.3 Ride History (if implemented)

- Offers functionality to view past rides

### 4.3 Load Balancing

The rider client implements client-side load balancing:

- Maintains a list of available server instances (discovered via etcd)
- Supports multiple load balancing policies (e.g., round-robin, pick-first)
- Automatically routes requests to healthy server instances

### 4.4 User Interface

Similar to the driver client, the rider client uses a command-line interface:

- Simple menu for ride requests and status checks
- Clear display of ride information and status updates
- Easy-to-follow prompts for entering ride details

### 4.5 Error Handling

The client includes comprehensive error handling:

- Graceful handling of server errors or unavailability
- Clear communication of any issues to the user
- Retry mechanisms for failed requests

## 5. Shared Components and Protocols

### 5.1 Protocol Buffers

Protocol Buffers play a crucial role in defining the communication protocol between clients and servers:

- **Service Definitions**: Clear definitions of RiderService and DriverService
- **Message Structures**: Well-defined structures for requests and responses
- **Code Generation**: Automatic generation of client and server code stubs

### 5.2 TLS and Security

Both clients and the server use TLS for secure communication:

- **Certificate Management**: Proper handling of CA certificates, server certificates, and client certificates
- **Mutual Authentication**: Both clients and servers authenticate each other
- **Encryption**: All communications are encrypted to protect user data

### 5.3 etcd Integration

etcd is used across the system for various purposes:

- **Service Discovery**: Clients discover server instances through etcd
- **State Management**: Server instances use etcd to maintain shared state
- **Distributed Locking**: Ensures data consistency in a distributed environment

# 6. Advanced Features and Considerations

## 6.1 Scalability

The system is designed with scalability in mind:

- **Horizontal Scaling**: Additional server instances can be easily added to handle increased load
- **Stateless Servers**: Core application logic is stateless, with state managed in etcd
- **Efficient Communication**: Use of gRPC and Protocol Buffers ensures efficient data transfer

## 6.2 Fault Tolerance

Several measures are in place to ensure system reliability:

- **Redundancy**: Multiple server instances can run concurrently
- **Automatic Failover**: Clients can switch to healthy servers if one becomes unavailable (through load balancing)
- **Data Persistence**: Critical data is stored in etcd, surviving individual server failures

## 6.3 Monitoring and Logging

The system includes comprehensive monitoring and logging:

- **Request Logging**: All requests and responses are logged
- **Error Tracking**: Errors are logged with appropriate context for easy debugging
- **Performance Metrics**: Key performance indicators are tracked (e.g., request latency, success rates)

## 6.4 Extensibility

The system is designed to be easily extensible:

- **Modular Design**: Clear separation of concerns between components
- **Versioned APIs**: Protocol Buffers support versioning for backward compatibility
- **Pluggable Components**: Key components (e.g., load balancing, authentication) are designed to be replaceable

# 7. Conclusion

The MyUber application is thus a well-architected, distributed system for ride-sharing. By leveraging modern technologies like gRPC, etcd, and Go, it provides a scalable, efficient, and secure platform for connecting riders with drivers. The modular design and use of industry-standard protocols ensure that the system

can evolve to meet future requirements while maintaining robustness and reliability.

This comprehensive approach to system design, coupled with careful consideration of security, scalability, and user experience, positions MyUber as a solid foundation for a production-ready ride-sharing service. The clear separation of concerns between server, driver client, and rider client, along with the use of a distributed architecture, allows for independent scaling and enhancement of each component, making it well-suited for handling the dynamic and growing demands of modern transportation networks.