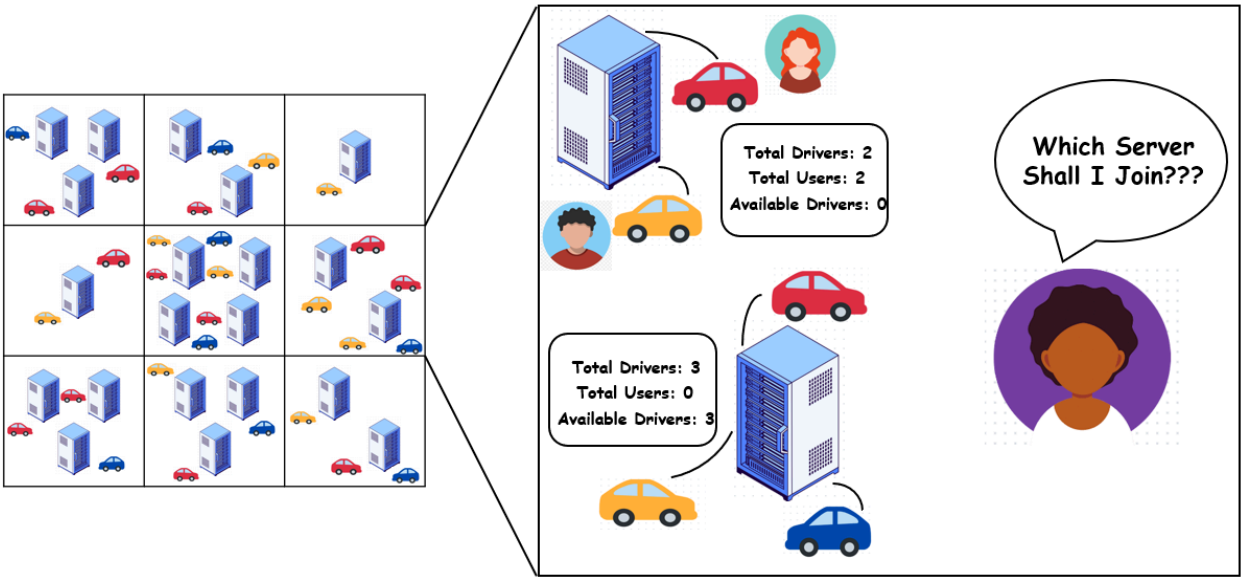
QuickRide using gRPC



Flow:

1. **Get Server Status:** Clients know the status of servers and the number of available drivers.
2. **Connection and Certificate Verification:** Clients connect to the selected server and undergo certificate checking to ensure secure communication, verifying the server's authenticity and establishing a secure data exchange channel.
3. **Ride Request and Waiting:** Once connected, drivers wait for ride requests, while riders can submit requests for rides. The server manages these requests and matches riders with available drivers.
4. **Driver Response to Ride Requests:** Upon receiving a ride request, drivers have a 10-second timer to accept or reject it. If a driver accepts the request, the ride is confirmed, and the driver is notified.
5. **Ride Completion:** After completing a ride, drivers can mark the ride as complete in the system, updating their status and availability for new

ride requests. This action helps maintain accurate data for both riders and drivers.

Assumptions

1. **Geographic Server Distribution**: The system is designed with multiple servers distributed across geographic areas. Each server manages requests specific to its region, optimizing response times and reducing latency for clients.
2. **Client Knowledge of Servers**: Clients (both riders and drivers) are assumed to know the port numbers for each server. This allows them to directly send requests for driver availability to the appropriate server without needing intermediaries.

Load Balancing Logic

1. **Rider Behavior**: Riders will connect to the server that has the highest number of available drivers. If the number of available drivers exceeds a specified threshold, riders can join this server directly without needing to check the status of other servers.
2. **Driver Behavior**: Drivers will connect to the server with the fewest available drivers. If the number of available drivers falls below a certain threshold, drivers can join this server directly without assessing the status of other servers.

Overview of API Methods :

The QuickRide API includes various methods for both riders and drivers, facilitating seamless interaction and efficient ride management. Below is a detailed breakdown of the implemented methods:

Common Methods for Both Riders and Drivers:

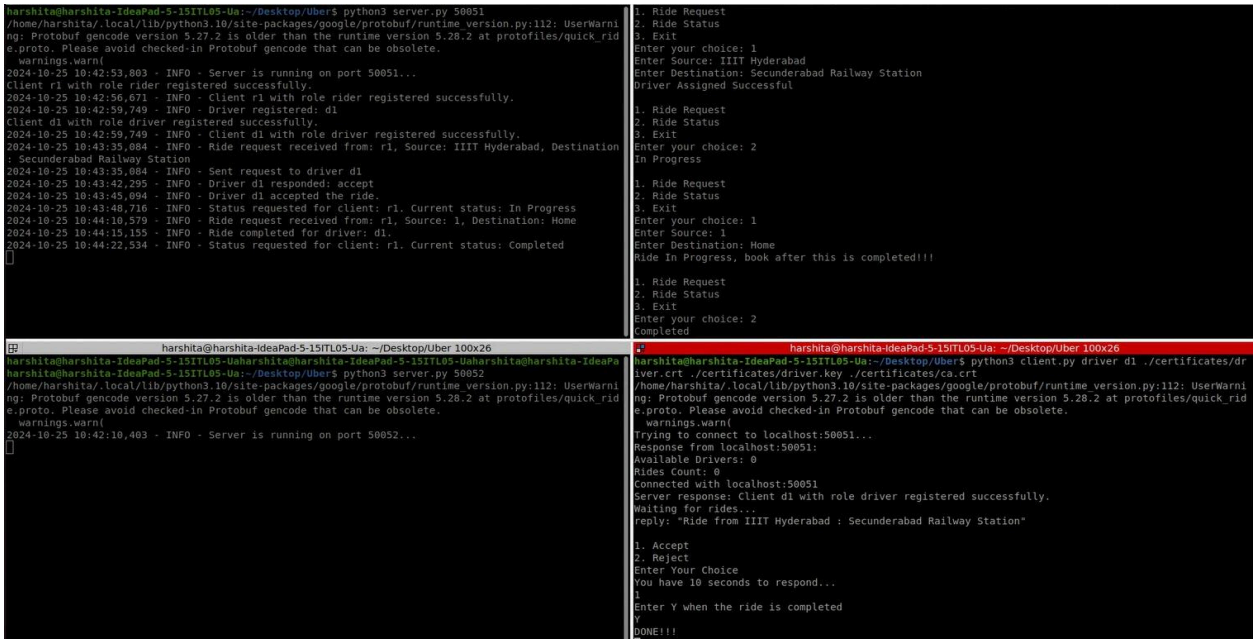
- **`rpc ServerStatus (ServerStatusRequest) returns (ServerStatusResponse)`**
  Clients can query the status of the server to determine its availability. This allows them to make informed decisions about whether to connect to the server.
- **`rpc RegisterClient(RegisterRequest) returns (RegisterResponse)`**
  Clients can register with the server using a unique identifier. This process ensures that each client is recognized and can participate in the ride-sharing service.

Rider-Specific Methods:

- **`RequestRide(RideRequest) returns (RideResponse)`**
  Riders initiate a ride request by specifying their pickup and destination locations. If a driver is available, they receive a notification to accept the ride. Upon acceptance, the ride status is updated accordingly.
- **`RideStatus(StatusRequest) returns (StatusResponse)`**
  This method allows riders to check the current status of their ride, providing real-time updates and enhancing the user experience.

Driver-Specific Methods:

- **`DriverNotify(DriverRequest) returns (stream DriverResponse)`**
  Drivers receive notifications about ride requests in real-time. If they are available, they can respond to these requests immediately.
- **`SendRideResponse(RideAccept) returns (RideAcceptMessage)`**
  Drivers have a 10-second window to either accept or reject a ride request. If they do not respond within this timeframe, the request is automatically considered rejected.
- **`CompleteRide(RideComplete) returns (CompleteReply)`**
  After accepting a ride, drivers can mark the ride as completed, finalizing the transaction and updating the system accordingly.



Scope of Improvements :

Lack of Server Communication: The current implementation does not allow servers to communicate with each other. To enhance the system, a communication protocol can be established between servers, enabling them to share real-time data about available drivers and ride requests.

Data Storage for Client and Ride Details: There is currently no centralized storage for client and ride information. Implementing a database would provide a structured way to store and manage client profiles, ride history, and driver availability, facilitating better data retrieval and analysis.

Handling Cross-Region Rides: The system lacks functionality for managing rides that start in one geographic region and end in another. By implementing a method for data sharing between servers in different regions, the system can ensure a seamless experience for riders and drivers during cross-region trips.