Report on MyUber Question

October 21, 2024

1 Ride-Sharing Service APIs

The ride-sharing service is implemented using gRPC, providing an interface for clients (riders and drivers) to interact with the service. It includes the following API methods:

- **RequestRide**: This method allows a rider to request a ride. It generates a unique `ride_id`, stores the ride details, and attempts to assign an available driver. If no driver is available, the system informs the rider.
- **RegisterDriver**: This allows drivers to register themselves and mark their availability.
- **AssignRide**: Assigns rides to drivers and helps manage the current rides.
- **GetRideStatus**: A method to check the status of a ride based on the `ride_id`.
- **AcceptRide/RejectRide**: Drivers use these methods to accept or reject a ride. A rejection triggers re-assignment.
- **CompleteRide**: Marks the ride as completed by the driver and updates the driver's availability.
- **GetAssignedRide**: This allows the driver to check if any rides are assigned.

The gRPC service is structured with threading to handle multiple requests concurrently and uses unique identifiers (`ride_id`, `driver_id`) to track the status of ongoing rides.

2 SSL/TLS-Based Authentication

The server and client communication is secured using mutual TLS (mTLS) authentication. Both client and server must present valid certificates for successful communication. This ensures the authenticity of both parties:

- **Server-side**: SSL certificates are loaded using `grpc.ssl_server_credentials`, which include the server certificate (`server.crt`), private key (`server.key`), and the certificate authority (CA) file (`ca.crt`).
- **Client-side**: Each client (driver, rider) also loads their own SSL certificates (`rider_client.crt` or `driver_client.crt`), private key, and CA certificates, ensuring secure communication with the server via `grpc.ssl_channel_credentials`.

This implementation guarantees secure communication and prevents unauthorized access.

3 Authentication And Logging Interceptor

The system uses an authentication interceptor (`LoggingInterceptor`) for logging requests and responses. Although primarily designed for logging, it can also serve as an authentication mechanism by validating client roles (rider/driver). The interceptor:

- Captures method names, timestamps, and responses.
- Appends log entries to a log file.
- Ensures that all client calls are logged, providing traceability and insights into request handling.

The interceptor is used across various client-side and server-side communications to maintain consistency in logging activities.

The `LoggingInterceptor` class is used to intercept all client requests (rider and driver) and server responses. It logs the method names, requests, and responses to a log file (`log.txt`), providing detailed insights into interactions between clients and the server. This interceptor ensures that every request and response is logged with timestamps, method names, and response details.

Key functionalities of the interceptor:

- Logs each gRPC method call with the method name and role (rider/driver).
- Logs responses from the server, capturing the complete lifecycle of a request.

4 Timeout and Rejection Handling

Timeouts and rejection handling are implemented to ensure that unresponsive drivers or rejected rides are handled efficiently:

- **Timeout Handling**: Once a driver is assigned to a ride, a timeout thread is started. If the driver doesn't respond within the timeout period (10 seconds), the system marks the ride as timed out and attempts to reassign it to another available driver.

- **Rejection Handling**: If a driver rejects a ride, the ride is placed back into a queue and reassigned to another available driver, while keeping track of the drivers who rejected the ride (to avoid reassigning it to them).

This mechanism ensures that unassigned or rejected rides are handled seamlessly, improving the reliability of the system.

5 Driver Availability and Ride Assignment

The server keeps track of available drivers using a dictionary (`self.drivers`). When a rider requests a ride, the server looks for an available driver:

- **Assigning Rides**: The server checks for drivers with the status ‘available‘ and assigns the ride. If no driver is available, it sends a response indicating that no drivers are available.
- **Reassigning Rides**: In case of rejection or timeout, the ride is reassigned to another driver, ensuring that rides are completed without delays.

This system is crucial for dynamic ride allocation, ensuring efficient matching between riders and drivers.

6 Client-Side Load Balancing

A client-side load balancer is implemented to distribute rider and driver requests across multiple ride-sharing service servers:

- **Rider Load Balancing**: The rider sends requests to multiple servers sequentially (based on the list provided by the load balancer) until a server responds with an available driver.
- **Driver Load Balancing**: The load balancer assigns drivers to the server with the least load (based on the number of drivers registered on each server), ensuring balanced distribution of drivers across multiple servers.

This mechanism ensures that requests are distributed evenly across the available servers, preventing server overload and optimizing resource usage.

7 Conclusion

In conclusion, the ride-sharing system is a scalable, secure, and highly efficient service. It implements essential features such as secure authentication (mTLS), logging, timeout/rejection handling, and load balancing, making it well-suited for real-world applications where dynamic allocation and secure communication are crucial.