Distributed Systems

Homework-4 Question-3 Report

Kritin Maddireddy (2022101071)

December 31, 2024

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

MyUber

Implementation of the required functionalities

General details

Firstly, the *server* is *asynchronous* as it would need to communicate with multiple clients at once without blocking for any one client. The *client*, however, is *synchronous*, as they will communicate with at most 1 server at any given time.

When each client/server starts up, a Unique User ID (UUID) is generated for both the client and the server, and this UUID is used to identify the client/server.

SSL/TLS-Based Authentication

In order to encrypt all communication between the client and the server, I initially created a CA certificate, and a bash script to generate SSL certificates on demand.

When a client/server starts up, it calls this bash script to generate an SSL certificate for itself, and uses the CA certificate to sign this certificate. The certificate and the private key are passed as `SslCredentials`, which are then used for communication.

Every single time, every single client/server has a unique certificate, which essentially is a one-time use certificate (by virtue of the way in which UUIDs are randomly generated).

Riders and Drivers are differentiated by having their role specified (along with their UUID) in the CommonName (CN) of the certificate, while the server they're connecting to is specified in the SubjectAlternativeName (SAN).

Interceptors

Both the Authorization and Logging interceptors, as well as the Additional Validation interceptor are implemented in a single interceptor class, called the `LogAndAuthInterceptor`.

When a gRPC call first arrives, the server obtains the role of the client, the timestamp at which this call was made, the expiry date of the certificate (for additional validation), and the method that was called. All of these are asynchronously logged into a log file present at `.log` within the server directory.

Locks are used to prevent race conditions when each server logs their gRPC request to the `.log` file. However, this is an issue only when multiple servers run on the same machine. If they run on different machines as they're intended to be, then no issue occurs.

Using this data obtained previously, the server then either allows the user access to the call, or aborts the call if they haven't provided a certificate, or if their certificate has expired, or if their role and the call they're making is mismatched.

Timeout and Rejection Handling

A timeout of 10 seconds is defined for every request, which is imposed by the server. If the driver doesn't respond within 10 seconds, then the server automatically reassigns the request to another available driver. The same happens when a driver rejects this request.

If no drivers are available after at most 3 reassignments, then the request is cancelled. However, as a singular rider can request for multiple rides, I cannot keep a stream open from a server to the rider, to notify them of the cancellation immediately. Alternatively, when the user checks the status of their ride, they can see that it is cancelled.

Now, on the driver's end, because waiting for user input is a blocking statement, I cannot automatically cancel the request when it times out. However, when the driver eventually responds, if they reject it's all good since the server will have already reassigned the request anyway. However, if they accept, then the server has to respond again with an acceptance confirmation (`ACCEPTANCE` status), before the driver can begin the ride. If the acceptance happens

after the timeout of the request, then the server responds with a `TIMED\_OUT` status, to let the driver know that they've responded too late.

Driver Availability and Ride Assignment

Driver's status is stored in the Redis in-memory database that I'm using.

The server that first receives the request from a rider will choose an available driver (one who has their status set to `available`) to assign this request to, and update their status to `on_ride` in the Redis database, and publishes this assignment to the `RideRequestChannel`. Each server is subscribed to this channel, and when a ride request is received, it checks if the request is for a driver that has an open bi-directional streaming channel with this server. If it does, then the ride request is forwarded to the driver, and a 10 second timeout is set.

If the driver doesn't respond to the request, or if they reject the request, then that driver's status is set back to `available`, and they're added to the list of drivers who rejected this ride request. Then, the same server searches for another available driver who hasn't previously rejected this request, and sets their status to `on_ride` and publishes this assignment to the `RideRequestChannel`. The same process is repeated until either some driver accepts, or until the 3 driver rejection limit is reached.

If a driver accepts, then once they mark the ride as completed, their status is updated in the Redis database to `available` so that they can start accepting new requests.

Load Balancing

In order for the client to start up, the user needs to provide the load balancing policy that they'd like to use (one of `pick_first` or `round_robin`), along with the number of servers that they'd like to use, and the ports of all the servers that they'd like to connect to.

Then, these details are used to set the load balancing policy in `ChannelArguments` of gRPC, using which a `CustomChannel` is created, and all the ports are used to create an address of the form `ipv4:ip1:port1,ip2:port2,ip3:port3`.

Following this, gRPC is responsible for handling the load balancing, by analysing the server availability and load metrics. For `pick_first`, the first *available* server is picked, and for `round_robin`, each request is made to a different *available* server in Round Robin format.

Unfortunately, since my client is implemented in C++, I could not implement a custom load balancer that could actually make use of the server availability and load metrics, as C++ is not supported for Custom Load Balancers, according to this official gRPC website.

In order to implement Load Balancing, I had to use Redis to communicate with different servers, and maintain a common state across them, for drivers and ride requests.

A bi-directional stream between the driver and a server (as determined by the load balancing protocol) is open for the duration of a single ride request being sent to the driver. If the driver accepts/rejects or if the request times out, this stream is closed and the driver opens a new stream with a different server.