MyUber - gRPC

Distributed Systems : Assignment 4 - Q3

October 21, 2024

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

1 Introduction

The implementation prioritises *complete autonomy of server instances*. Each server maintains its own, unique, state. Thus, the state is completely distributed, the only point of homogeneity being the fact that all of them write to the same file on launching to denote active servers.

The major assumption of this method is the *lack of server crashes*.

2 High-Level Implementation

2.1 Server State

Each server maintains state for each request it receives. On receiving a *ride request*, it returns a globally unique *ride-id* to the rider. It stores a **map** from this *ride-id* to metadata about the ride, including the *status*, the *number of reassignments*, the *start and end locations* and so on.

It also maintains a **queue** of *ride-id*'s denoting the order in which rides will be offered to the driver-assignment-requests, to ensure fairness. If a ride hits a timeout or is rejected, it moves to the back of the queue and its metadata is updated.

2.2 The Client

The clients make requests to servers based on the load-balancing policy of choice. If a request involving a particular ride-id is made to a server that does not contain this ride-id (note that it is globally unique), then the client must simply try again until it finds the right server. This is a consequence of complete autonomy.

Ride requests are persisted in memory, while driver details are not. In order to express interest in taking a ride, the driver must explicitly request the server to assign them one. If there is no ride-request pending on the current server, the driver must try again.

3 Low Level Implementation - API's

3.1 RiderService

- **RequestRide** - A rider sends details about the ride it desires (name, source and destination) to the server. If the ride request is stored successfully, a globally unique ride-id is returned that is to accompany future requests involving this ride.

- **GetStatus** - A rider sends the *ride-id* of the ride it wishes to know about. It will get a status response:
  - PENDING: the ride is in the waiting queue
  - ASSIGNED: the ride has been assigned to a driver, who has neither rejected nor accepted it yet
  - ACCEPTED: the driver has accepted the ride, the ride is *ongoing*
  - COMPLETED: the driver has denoted that the ride is completefd
  - CANCELLED: the ride has been cancelled due to too many re-assignments

  along with details such as the driver (if accepted or completed) and the number of re-assignments so far.

3.2 DriverService

- **AssignDriver** - A driver, if available, sends a request to the server to be assigned a ride. The server pops from its queue, and sends the details of the ride to the driver as a response.
- **AcceptRideRequest** - the status of the ride changes to ACCEPTED, and it is not re-added to the queue.
- **RejectRideRequest** - the number of re-assignments gets incremented.
  - *if the number of reassignments is less than the permitted number*: the status of the ride changes to PENDING, and it is pushed to the queue again.
  - *else*: the status of the ride changes to CANCELLED, and it is not pushed to the queue again.
- **TimeoutRideRequest** - functions essentially the same as RejectRideRequest. The reason for a different API is that earlier, I was not incrementing the number of re-assignments in case of timeout, but now I am because of cases such as network faults (if timeouts keep occurring because of network faults, the ride request should also be cancelled eventually, lest the rider stay waiting forever).
- **CompleteRideRequest** - change the status of the ride to COMPLETED. It is not re-added to the queue.

Note that for all of the API's beyond the first in both services: the client will have to keep re-attempting the request until it finds the server on which the ride's state is stored.

4 Other Features

4.1 Time-Out

The timeout is handled on the client side. If the driver takes longer than some fixed duration to respond to an offered ride, then the client queries whether the user is still present, and sends a *TimeoutRideRequest* to the server so a ride can be re-assigned.

4.2 Authentication

I create separate data for the server and the clients (rider and driver). On the client side, I:

- load the certificate of the CA (certificate authority) who signed *the server's certificate* and append it to a certificate pool.
- load the client's certificate and private key.
- create credentials from the above

On the server side:

- load the certificate of the CA (certificate authority) who signed *the client's certificate* and append it to a certificate pool.
- load the server's certificate and private key.
- create credentials from the above.

When a client connects to a server, mutual TLS will take place (automatically, based on the credentials), in which:

- The client presents its certificate to the server
- the server verifies the client's certificate by checking the certificate chain against the CA certificate pool loaded earlier
- the server presents its certificate to the client
- the client verifies the servers certificate by checking the certificate chain aginst the CA certificate pool loaded earlier.
- the private key of the client is used to sign data sent to the server and vice versa. The public key of the client (shared during mTLS) is used to verify this data.

Additionally, there is an *authentication-interceptor*, which verifies that the client has been authenticated. Also, it reads the common name to ensure that only

a rider sends RiderService requests, and only a driver sends DriverService requests.

The certificate is also used by a *metadata-interceptor*, which logs the province and country of the client.

4.3 Load Balancing

I implemented a custom resolver based on the official examples. The *resolver* is responsible for resolving some name-space into a set of addresses the client can communicate to. It stores a number of ports where the server is serving certain services, and the load-balancer picks among them based on its policy.

While *pick_first* and *round_robin* are in-built, I implemented a custom load balancer called **random_picker** that selects a random sub-connection to route the rpc to. This is not necessarily practical, but was implemented as an exercise.

For my particular system, *round_robin* is often infeasible because of the autonomy of servers. Say a rider registers a request and immediately calls *GetStatus*, this request may be routed to every subsequent server before we finally hit back at the server storing the required *ride-id* and serve the request.

Thus, if we have a small system (less drivers and riders), *pick-first* is optimal. The requests are never routed to the wrong server and will be efficiently handled. If we have a larger system, *random-picker* may be better than *round-robin* as we may not need to cycle through all processes to serve the client. Although, there is a probable chance of a higher number of incorrect-server requests, and it is possible (theoretically) to never hit the correct server.

5 Enhancements and Future Considerations

The data access to the server-state is already protected by a mutex, in order to facilitate an easy-shift to a multi-threaded server, enabling us to serve the clients faster.

The globally unique ride-id is created as *<server-port-num:ride-num>*. This means the client can deterministically determine which server to query based on the id it is allotted. If we carry on with the assumption of minimal crashes and partitions, it will be most optimal to use this to query the exact server storing our state.

Alternatively, we may try to load-balance in such a manner as to distribute the initial port that a client connects to, instead of distributing the rpc's. This would ensure that the client always queries the same server for optimality, while

still distributing the ride-requests made. In such a system, the driver would keep cycling through the servers until it gets a ride it requires.

Frankly, this above idea is how I was implementing my system, until I came to the realisation that the balancer in grpc-go distributes the rpc's across the servers.

5.1 Driver Availability

Based on the system, driver availability need not be explicitly accounted for. A server will not assign a driver unless the driver pings the server saying it is available.