gRPC Assignment

Deadline: OCT 14 2024, 11:59 PM

Introduction

In this assignment, you will construct distributed solutions to certain problems and implement them using the gRPC(gRPC Remote Procedure Calls)

gRPC is an open-source remote procedure call (RPC) framework developed by Google that uses HTTP/2 for transport and Protocol Buffers for interface definition. It enables efficient, strongly typed communication between client and server applications, supporting features like authentication, bidirectional streaming, and flow control. GRPC simplifies building scalable, distributed systems and microservices by providing a unified communication mechanism.

You can use any programming language of your choice (**C/C++/Java/Python/Go/NodeJS** is strongly recommended owing to the amount of documentation available online).

- Documentation:
  - General: https://grpc.io/docs/
  - c++: https://grpc.io/docs/languages/cpp/basics/
  - Python: https://grpc.io/docs/languages/python/
  - go: https://grpc.io/docs/languages/go/basics/
  - protobuf: https://protobuf.dev/overview/
  - nodejs: https://grpc.io/docs/languages/node/basics/
- Resources:
  - https://github.com/grpc-ecosystem/awesome-grpc

Grading

- Teams from Assignment 1 will continue for Assignment 2
- **If one person doesn't work at all, then that person will get a negative score equal to the score got by the other person**. So please do your part in the assignment. (Not doing assignment gets a person a negative not a zero)
- This assignment will have manual evaluations, where you will do a demo of your solutions, and then questions will be asked about:
  - Concepts tested in the assignment (gRPC, protobuf, streaming, authentication, load balancing, interceptors among others)
  - Implementations for the given problem (you could be asked about parts which your teammate programmed also)
- **You can take your own reasonable assumptions where things aren't mentioned**

1 Labyrinth (25 Points)

In this problem, you are required to implement a single-server single-client architecture for the Labyrinth game.

A Labyrinth is simply an $m \times n$ grid containing different types of tiles. In our version, there are 3 types of tiles possible:

- *Empty tile*. As the name suggests there is nothing on this tile.
- *Coin tile*. This tile contains a coin. After the coin is collected, this tile will turn into an empty tile.
- *Wall tile*. This tile contains a wall and it is not possible to visit this tile. This tile will be converted to an empty tile if the wall is destroyed.

The player will start at the top left corner of the Labyrinth and the goal is to reach the bottom right corner while collecting as many coins as possible. In one move, the player can either move one tile or perform a spell. The player can possibly move in 4 directions, up, down, left or right. One of three things can happen:

- The player moved into an empty tile.
- The player moved into a tile with a coin. The coin will be collected and the tile will become empty.
- The player moved into a wall. In this case, a collision will happen and the player will lose a health point.

The player may also perform one of 2 types of spells:

- *Revelio*. This spell will be performed on a target tile with a tile type in mind. All the surrounding tiles (consider the 3×3 square around the target) with the desired tile type (coin/wall) will be revealed.
- *Bombarda*. This spell will destroy all objects on 3 tiles of the player's choice, converting these tiles to empty tiles.

Initially, the player has three health points and can perform a total of three spells.

1.1 Server

To initialize the game, the server will read a Labyrinth from a file and initialize all player attributes. From then on, the server needs to keep track of all the player attributes and Labyrinth updates: the player score, player health, remaining spells and the current position. A total of 5 RPCs need to be implemented.

- **GetLabyrinthInfo**: Take an empty request and provide the width and height of the Labyrinth.
- **GetPlayerStatus**: Take an empty request and provide the player's score, health points, current position in the Labyrinth and the number of remaining spells.
- **RegisterMove**: Take a request containing the desired direction to move and move the player in that direction. Provide the status of the move informing the client whether the move 1. was a success, 2. was a failure, 3. lead to victory or 4. resulted in player death.
- **Revelio**: Take a request containing the target position of the spell and the desired tile type. Provide a *stream* of positions of all the tiles surrounding the target tile having the same tile type as the one in the request. Note that the target tile is also included under the spell effect.
- **Bombarda**: Take a *stream* of target positions as input. Destroy anything that may be present on the target tiles (wall or coin), converting them into an empty tile.

The request and response objects for each method must contain the attributes mentioned in its description. You cannot change the method names, and streaming must be used wherever it is mentioned. However, the exact implementation of these objects is up to you. You may include additional fields if you feel it is necessary.

1.2 Client

On the client side, you need to demonstrate and prove that each of the RPC methods implemented on the server works as expected. Cover all cases that can be encountered in the game. The minimum requirement is to provide an interface to call each RPC method through the terminal.

1.3 Bonus

This section can recover the points you lost in **this problem only**. Create a terminal interface through which the Labyrinth game can be played.

2 K Nearest Neighbours (15 points)

The k-Nearest Neighbors (k-NN) algorithm is a instance-based machine learning algorithm used for classification and regression. In k-NN, the algorithm identifies the k data points in the training dataset that are closest to a given query point and makes predictions based on these neighbors.

Implement a distributed k-NN system where:

- A gRPC server holds different portions of the dataset.
- A client queries the server with a data point and the value of k to find the k-nearest neighbors.
- The client may need to contact multiple gRPC servers, each handling a different part of the dataset, and aggregate the results to find the global k nearest neighbors.

2.1 Components

1. gRPC Servers (Workers)
   - Each gRPC server holds a subset of the dataset.
   - When queried with a data point and k, it computes the k nearest neighbors within its subset and returns them to the client.
   - The dataset can be preloaded or provided as input (your choice).
2. gRPC Client (Master Node)
   - The client sends the query to multiple gRPC servers.
   - After receiving the k nearest neighbors from each server, it aggregates the results to find the global k nearest neighbors.
   - The client must handle communication and coordination between different servers.
3. Dataset

2.2 Task Breakdown

1. On the server-side, implement a gRPC service that provides the following RPC method:
   - FindKNearestNeighbors(request: KNNRequest) → KNNResponse
   - The request includes:
   - 'data_point': The query point for which neighbors are to be found.
   - k: The number of nearest neighbors.
   - The server returns the nearest neighbors (within its own subset of data).
2. On the client-side, implement a client that:
   - Splits the computation by contacting multiple gRPC servers.
   - Aggregates results from each server to determine the final k nearest neighbors.
   - The client should use efficient methods to collect, merge, and rank the results.
3. Dataset Partitioning: Split the dataset into different parts and distribute them across multiple gRPC servers.
4. Distance Calculation: Use Euclidean distance as the metric for calculating the nearest neighbors.
5. Comparative Analysis (keep it brief)

- Compare how this differs from MPI-based computation (e.g., how MPI uses tightly-coupled message passing).

6. Efficiency Considerations (**Bonus**): Handle communication efficiently using gRPC's streaming features where appropriate (e.g., streaming partial results). If you attempt this, mention relevant details in the report.

2.3 Deliverables

2.3.1 Code

- gRPC service and client implementations.
- Dataset partitioning and distance calculation.

2.3.2 README.md

- Document how to run and check your program

2.3.3 Report

- Discuss how gRPC facilitates distributed computing in this example.
- Comparison between gRPC and MPI in terms of communication models, usability, and scalability.
- Performance analysis for different sizes of datasets and k.

3 MyUber (35 Points)

Objective:

Design and implement a distributed ride-sharing platform using **gRPC** with **SSL/TLS** for secure communication and authentication. The platform should support **load balancing** to distribute requests across multiple servers. The system should handle ride requests, ride acceptance, and ride completion while ensuring that drivers and riders are securely authenticated using SSL/TLS certificates. Additionally, the system should manage **timeout**, **rejection**, and **load balancing** across multiple servers.

Problem Requirements:

3.1 Ride-Sharing Service API

The following API methods should be implemented:

- Riders:
  - **RequestRide(RideRequest)** → **RideResponse**: Riders can request a ride by specifying their pickup location and destination. The system will attempt to assign a driver, considering timeout and rejection handling.
  - **GetRideStatus(RideStatusRequest)** → **RideStatusResponse**: Riders can check the status of their ongoing or requested rides.
- Driver:
  - **AcceptRide(AcceptRideRequest)** → **AcceptRideResponse**: Drivers can accept ride requests assigned to them. If a driver does not respond within a certain timeout period or explicitly rejects the ride, the ride will be reassigned.
  - **RejectRide(RejectRideRequest)** → **RejectRideResponse**: Drivers can explicitly reject ride requests if they are unable or unwilling to accept them. The system will automatically reassign the ride to another available driver.
  - **CompleteRide(RideCompletionRequest)** → **RideCompletionResponse**: After dropping off the rider, drivers can mark the ride as complete.

3.2 SSL/TLS-Based Authentication

- All communication between the client and the server must be encrypted using **SSL/TLS**.
- **Mutual TLS (mTLS)** should be implemented for both server and client authentication, ensuring that clients (riders and drivers) provide valid certificates issued by a trusted CA.
- **Client Certificates**:
  - Riders: The system should accept client certificates issued for riders.
  - Drivers: The system should accept client certificates issued for drivers.
- **Server Certificate**: The server will provide clients with a certificate for secure communication.

Authencation Documentation: https://grpc.io/docs/guides/auth/

3.3 Interceptors

- **Authorization Interceptor**: Implement an interceptor that verifies the client certificate to determine whether the client is a driver or rider. This ensures that only **riders** can request rides, and only **drivers** can accept or complete rides.

- **Logging Interceptor**: Log all gRPC calls, including method names, timestamps, and client roles (driver or rider).
- **Additional Validation**: The interceptor can check additional metadata from the client certificate, such as the certificate's expiration date or the role of the user.

Interceptors Documentation: https://grpc.io/docs/guides/interceptors/

3.4 Timeout and Rejection Handling

- Drivers must either **accept** or **reject** a ride within a specified timeout period (e.g., 10 seconds).
- If a driver does not respond within the timeout period, the system will automatically **reassign** the ride to another available driver.
- If a driver explicitly **rejects** a ride, the system will attempt to reassign the ride to another available driver.
- If no drivers are available after multiple reassignments, the ride request may be **canceled**, and the rider will be notified.

Deadlines Documentation (If needed): https://grpc.io/docs/guides/deadlines/

3.5 Driver Availability and Ride Assignment

- **Drivers who are currently on a ride** should not receive new ride requests until they have completed their current ride.
- Drivers must update their status to **available** after completing a ride to receive new ride requests.

3.6 Load Balancing

- Implement **client-side load balancing** using the algorithm across multiple gRPC servers.
  - **pick_first**
  - **round-robin**
  - **Custom Load Balancing** Implement Your own custom load balancing. (**This is a Bonus**, this can used to gain points lost in this Question).
- The client should distribute ride requests across multiple servers (e.g., on ports 50051, 50052, 50053) transparently.
- Optionally, integrate with a **service discovery** system (e.g., DNS, Consul, etcd) for dynamically discovering available servers.

Load Balancing Documentation Blog: https://grpc.io/blog/grpc-load-balancing/

Custom Load Balancing Documentation : https://grpc.io/docs/guides/custom-load-balancing/

File Requirements

Certificates and Keys

You will need the following certificates and keys for SSL/TLS:

Example Files:

- **ca.crt**: The Certificate Authority (CA) certificate used to sign the server and client certificates.
- **server.crt** and **server.key**: The server's certificate and private key.

- **client.crt** and **client.key**: The client's certificate and private key (for mutual TLS).

Proto File

Define the gRPC service and messages in proto file(s).

Server Files

Example:

- **ride_sharing_server.py**: Implements the gRPC service with SSL/TLS and interceptor logic.
- **auth_interceptor.py** (optional): Implements authorization logic based on the client's certificate.

Client Files

Example:

- **ride_sharing_client.py**: Implements the client-side code, which communicates securely with the server and distributes requests using round-robin load balancing.

ReadMe and Report

- **README.md**: Document how to run and check your program.
- **Report.pdf**: Explain How you have implemented the required Functionalities, and optionally bench marking of Different load balancing policies

4 Real-time Document (25 Points)

A real-time document is one where many collaborators can open the same document and change it, with the changes propagated to all other clients viewing the document. in all instances of the open document.

This problem requires an implementation of a real-time collaborative document with **gRPC** calls.

4.1 Objectives

- Document must be visible and available to modify using an appropriate interface in clients (terminal, web browser, etc.)
- Adding, editing and deleting text must be supported.
- The changes made by all clients are forwarded by a server to a dedicated logging program (can be as simple as modifying a text file) using a single direction asynchronous gRPC call.
- The changes made by one client are synced to another client through the server. This must be implemented with bidirectional asynchronous gRPC calls.
- Error handling:
  - In case of gRPC failure, attempt to reload the document or display an appropriate error message.
- Consistency is **required** except for the case where multiple collaborators attempt to modify the same location in the file.

There will be at least 4 separate programs running:

- Two or more clients, acting as an interface for users.
- One server, which syncs the text between clients.
- A database saves the stream of changes to be re-opened later.

The implementation can happen in steps:

1. Client sends changes to server.
2. There is one client, which sends changes to one server, which in turn sends changes to a logging program.
3. There are multiple clients and all must be synced.

You may add additional gRPC calls to implement this problem as you see fit.

4.2 Submission Requirements

Protobuf files

Define the messages and services in protobuf format.

Build command

Place **all** commands for building your program in a bash file named **toBuild.sh**

README

Document implementation and its quirks.

Report

Explain the implementation in detail and show that the described functionality is met.

4.3 Notes for implementation

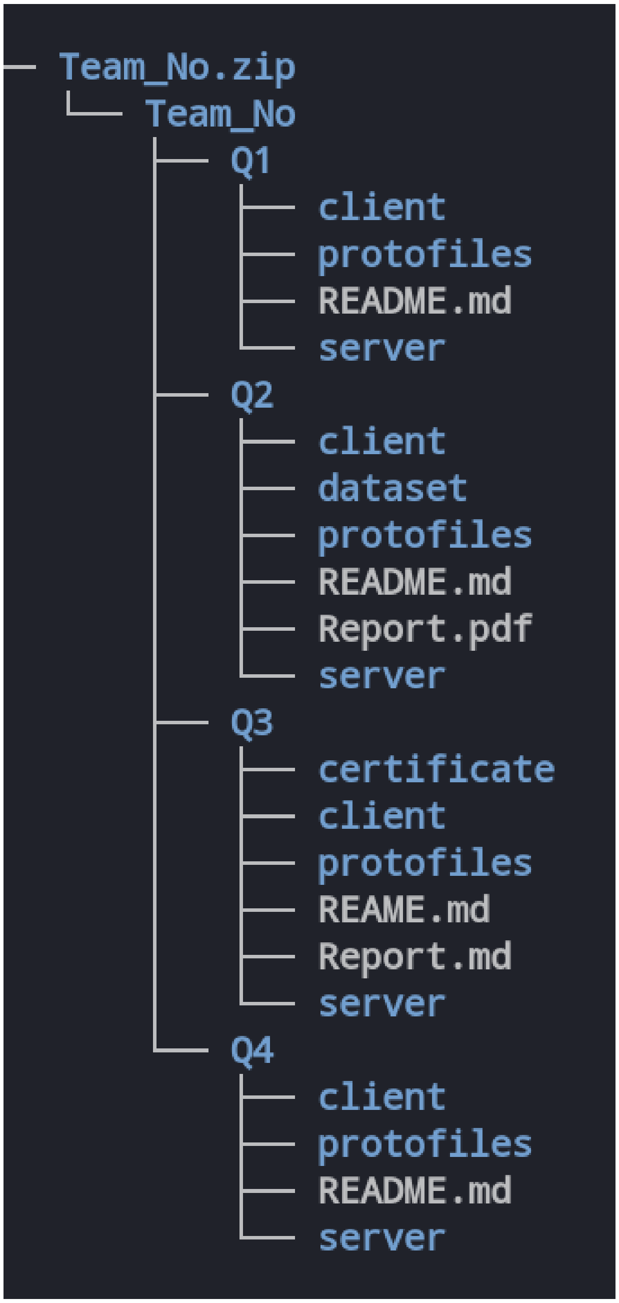
- You may mix and match front-end and back-end in languages according to convenience.
- **You may find it simpler to use python or javascript for this problem.**

5 Submission Format

We'll be manually checking the assignment with evals.

Your submission is expected to be a **<TeamNumber>.zip** file containing a directory with the same name as your Team number (ex: Team65)

A sample submission is shown in the diagram below.



```
─ Team_No.zip
  └── Team_No
      ├── Q1
      │   ├── client
      │   ├── protofiles
      │   ├── README.md
      │   └── server
      ├── Q2
      │   ├── client
      │   ├── dataset
      │   ├── protofiles
      │   ├── README.md
      │   ├── Report.pdf
      │   └── server
      ├── Q3
      │   ├── certificate
      │   ├── client
      │   ├── protofiles
      │   ├── REAME.md
      │   ├── Report.md
      │   └── server
      └── Q4
          ├── client
          ├── protofiles
          ├── README.md
          └── server
```