ECGR 4090/5090 Cloud Native Application Architecture
Lab 5: gRPC

In this lab we will develop a canonical gRPC application in Golang to get familiar with gRPC. Originally developed by Google, gRPC is an open source Cloud Native Computing Foundation (CNCF) graduated project. The project is available at https://github.com/grpc/grpc and is mostly written in C/C++. However, gRPC can be used from a number of different languages including Go, Python, Node, Java etc.
The material in the lab is mostly based on the official gRPC documentation available at grpc.io

gRPC is a modern, open source, high-performance remote procedure call (RPC) framework that can run anywhere. In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier to create distributed applications and services. As in many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub (referred to as just a client in some languages) that provides the same methods as the server. Note that the client and server stubs can be in different languages (for example client in Python and server in Go). In this lab, we will use Go for both the client and server.

Protocol Buffers

By default, gRPC uses Protocol Buffers, Google's mature open source mechanism for serializing structured data, typically in binary format, making for fast and efficient communication.

The first step when working with protocol buffers is to define the structure for the data to serialize in a proto file: this is an ordinary text file with a .proto extension. Protocol buffer data is structured as messages, where each message is a small logical record of information containing a series of name-value pairs called fields. Services are then built with the message types. For example,

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
```

}

The protocol buffer compiler, protoc, is used to generate data access classes in your preferred language(s) from the proto definition.

Installation

#Install the protoc compiler
$ sudo apt install -y protobuf-compiler
$ protoc --version # Ensure compiler version is 3+

Install the protocol compiler plugins for Go using the following commands:

$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
$ go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@v1.2

Ensure that your GOPATH was set correctly (done in Lab 1)
$ go env GOPATH

Update your PATH in your .bashrc so that protoc can find the plugins
$ export PATH="$PATH:$(go env GOPATH)/bin"

Run .bashrc
$ source ~/.bashrc

Check to see if installation worked correctly with the following sample implementation. In the *labs* directory,
$ git clone -b v1.61.0 --depth 1 https://github.com/grpc/grpc-go

$ cd grpc-go/examples/helloworld

Compile and execute the server code
$ go run greeter_server/main.go

In a separate terminal, run the client code. Should produce Greeting: Hello World
$ go run greeter_client/main.go

Congratulations! You've just run a client-server application with gRPC.

We'll now write a gPRC client-server application from scratch.

Client-server API with gRPC

We will develop a movie info service where clients can ask for information about a particular movie title, and the server responds with information for that particular movie (year, director, and cast).
Note: Please follow the directory structure described below to avoid missing paths

Under *labs/cloudnativecourse* create a *lab5* directory. The *cloudnativeclourse* directory was created in Lab 4

$ *mkdir lab5*

In the instructions below, replace *gitlab* by *github* if you are using github.

Next, in the *cloudnativecourse* directory, initialize the Go module pointing to your github/gitlab repository.

$ *go mod init gitlab.com/<username>/cloudnativecourse*
At this point, in the *cloudnativecourse* directory, you should see a *lab5* directory, and a *go.mod* file

Under the *labs/cloudnativecourse/lab5* directory, create a *movieapi* directory. Your directory structure now looks like lab5/movieapi
First define the structure of the gRPC message in a proto file. Download the *movieapi.proto* file from Canvas into the *movieapi* directory.

Add, commit, and push the local changes to your github/gitlab repository as described in Lab 4.

Note: Adjust the path in *go_package* to point to your directory (*gitlab.com/<username>...*)
cd to the *lab5* directory. The path specified on the Canvas point to my private Gitlab repo.

Compile the proto file
$ protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=.
--go-grpc_opt=paths=source_relative movieapi/movieapi.proto

In the *movieapi* directory, you will seen two new files - *movieapi_grpc.pb.go* and *movieapi.pb.go*
Examine *movie_grpc.pb.go* . This contains the server and client Go code code generated by *protoc* for the gRPC services defined in the *.proto* file
$ *cd ..*
Under lab5 create a *movieserver* directory. Download the *server.go* file from Canvas.

Note: Be sure to change the *movieapi* import paths to that appropriate to your system.

Under lab5 create a *movieclient* directory. Download the *client.go* file from Canvas.
Note: Change the *movieapi* import path to appropriate to your system.

Run the server
$ go run movieserver/server.go

If you get the following error - *undefined: grpc.SupportPackageIsVersion7 grpc.ServiceRegistrar* do the following -
$ go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
$ go get -u google.golang.org/grpc
Recompile proto with protoc

In another terminal, run the client
$ go run movieclient/client.go
The client makes a request to the movie "Pulp Fiction". The movie information is stored in a map data structure on the server. The server processes the client request, and returns the information associated with the movie.

This completes the demo of gRPC client-server

Note that gRPC also supports client, server, and bidirectional streaming. For examples see, https://grpc.io/docs/languages/go/basics/
Also, gRPC supports SSL/TLS integration, which can be used to authenticate the server (from the client's perspective), and to encrypt message exchanges.

Note: Be sure to update your Gitllab/Github repository if you haven't already done so.

To do -
Extend the *MovieInfo* gRPC service by adding a new rpc *SetMovieInfo* that takes as argument the message *MovieData*, and returns the message *Status*. The message *MovieData* includes fields such as title, year, director, and cast. The server stores this in *moviedb*. The message *Status* includes a field called code (string) indicating success or failure of operation. Test the service by printing debug messages, and by using the rpc *GetMovieInfo* to retrieve the newly inserted movie data

Hint: You will need to modify the proto file, generate client and server stubs with protoc, and modify server, and client code. Examine the *movie_grpc.pb.go* to check out the stubs that protoc generates. You will need to use these in the client and server code.