GRPC

Spring 2018

Contents

1 Introduction

This lab is intended to get you more familiar with gRPC which you will require to implement everything from scratch in Puddlestore. An excellent resource to learn gRPC is the official docs themselves. This lab will use much of the relevant portions from this Go tutorial. You can learn how to setup and install everything via this quickstart guide.

gRPC is Google's RPC framework and the protocol itself is built on http2. It has many advantages such as the ability to enable different languages to interact with each other via gRPC calls.

gRPC uses Protocol Buffers for serialization/deserialization of objects and data structures sent between clients and servers. Developers need to define their data structures in .proto files and auto generated code provides functions that allow for these structures to be serialized/deserialized.

In this lab you will look at a small example to understand the basics of both gRPC and Protocol Buffers. Once you have an understanding, we will ask you to implement a small (yet hopefully fun) task for you to apply your newly acquired knowledge. This is an optional lab to increase your understanding about gRPC and will not be graded.

2 Protocol Buffers

In order for Protocol Buffers to serialize/deserialize an object, that object must be defined as a message in a `.proto` file. Lets look at a small example taken from the google documentation before we dive any further:

```
message Person {
  string name = 1;
  int32 id = 2;  // Unique ID number for this person.
  string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phones = 4;
}

// Our address book file is just one of these.
message AddressBook {
  repeated Person people = 1;
}
```

A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including `bool`, `int32`, `float`, `double`, and `string`. You can also add further structure to your messages by using other message types as field types.

Say our code has a `Person` object which we wish to serialize. We would need to define and lay-out a `Person` message corresponding to that class in our .proto file so that the Protocol Buffers compiler can accordingly handle the object and provide us with a method to serialize and deserialize it.

In the above example, the `Person` message contains `PhoneNumber` messages, while the `AddressBook` message contains `Person` messages. You can even define message types nested inside other messages – as you can see, the `PhoneNumber` type is defined inside `Person`. You can also define enum types if you want one of your fields to have one of a predefined list of values – here you want to specify that a phone number can be one of `MOBILE`, `HOME`, or `WORK`.

The `= 1`, `= 2` markers on each element are unique "tags" that uniquely identify the field in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements.

A field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the Protocol Buffers. You can also think about repeated fields as dynamically sized arrays.

There is a lot more to Protocol Buffers that can be learned via the documentation. However this should be more than sufficient for PuddleStore purposes.

3 gRPC Example

Now that we have an understanding of what Protocol Buffers are, we can move onto understanding more about the working of gRPC.

Protocol Buffers are the central part of using gRPC! In order to use gRPC, there are three things you have not learned to do in CS 1380 yet: defining RPC services, creating the RPC calls, and representing the input/return types of these RPC calls as Protocol Buffer messages. Note that you have already worked on implementing new RPC calls in Tapestry and Raft.

Let's look at our very own Chord as an example for this! To define a Chord RPC service, we can simply do:

```
service ChordRPC {
    ....
}
```

Now that we have the service, let's say we want to make an RPC call for retrieving the predecessor Id of some node and we want to call it `GetPredecessorIdCaller`. We can define it in the ChordRPC service as follows:

```
service ChordRPC {
      rpc GetPredecessorIdCaller(RemoteNodeMsg) returns (IdReplyMsg);
}
```

Now there are a few things to note here: `rpc` is a keyword to let know that we are defining an RPC method. This is followed by the name of the RPC function. We than provide the input arguments to the functions which will be a Protocol Buffers message defined later on. In this case the message is called `RemoteNodeMsg`. `returns (IdReplyMsg)` indicates that the return type of this RPC calls will be another Protocol Buffer message called `IdReplyMsg`. It is important to note here that a gRPC function can have only one input argument and one return type! In fact it must have one input argument and one return type. Even if you don't require an input argument and/or a return type, you must provide one. If such is the case, you can just create a placeholder message called `Empty`, for example, and use it across all such cases.

Now all that is left for us to do is define the messages we specified as the argument and return for our rpc function as follows:

```
message RemoteNodeMsg {
      bytes Id = 1;
```

```
        string Addr = 2;
}

message IdReplyMsg {
        bytes Id = 1;
        string Addr = 2;
        bool Valid = 3;
}
```

Our chord.proto file will look like this:

```
syntax = "proto3";

package chord;

service ChordRPC {
        rpc GetPredecessorIdCaller(RemoteNodeMsg) returns (IdReplyMsg);
}

message RemoteNodeMsg {
        bytes Id = 1;
        string Addr = 2;
}

message IdReplyMsg {
        bytes Id = 1;
        string Addr = 2;
        bool Valid = 3;
}
```

Note that `syntax = "proto3"` is required at the start of your proto file to indicate the version of Protocol Buffers being used and the package is chord. Now if you take a look at the chord.proto which was provided with Chord, you can see that its the same thing except with all the other RPC methods and messages that the actual Chord implementation needs.

It is worth noting that there are many other variants of RPC calls that can be defined in a .proto file. These include server-side streaming, client-side streaming etc. You can learn more about these in the documentations.

Once we have our .proto file created, we need to convert it into something that we can use in our Go Code. That is where the magic comes in. Assuming our `.proto` file is in a folder call `Chord`, we can run the following command:

```
protoc -I Chord/ Chord/chord.proto --go_out=plugins=grpc:chord
```

The above command will take the `chord.proto` file and generate a file called `chord.pb.go`. This file will contain compiler generated code that we can simply plug into our Chord implementation and start filling out right away! The command `protoc` will take in an import folder specified via `-I Chord/` to scan for imports. `Chord/chord.proto` is the location of the .proto file to compile. `--go\_out=` means the language to compile our .pb.go is go. `plugins=grpc` indicates to use the grpc plugin and `:chord` is the output folder for our compiled .pb.go.

The main thing you need to take away from the generated file `chord.pb.go` is the interface you need to implement on the server side of your application. The functions of this interface are always of a standard form:

```
type ChordRPCServer interface {
      GetPredecessorIdCaller(context.Context, *RemoteNodeMsg) (*IdReplyMsg, error)
  }
```

The name of the interface will be whatever you called the service in the `.proto` file appended with the string `Server`. So in our example, since we called the service `ChordRPC`, the interface is called `ChordRPCServer`. The RPC function takes in the `RemoteNodeMsg` type and an additional `Context` parameter. Unless you're doing XTrace, you shouldn't be concerned with that parameter and can use `context.Background()` as a default input. The return types are self-explanatory.

There we have it! Now we simply need to implement the function `GetPredecessorIdCaller` in our code and we can use it right away.

4 gRPC Methods

To call gRPC methods, we first need to create a gRPC channel to communicate with the server. We create this by passing the server address and port number to `grpc.Dial()` as follows:

```
conn, err := grpc.Dial(*serverAddr)
if err != nil {
    ...
}
defer conn.Close()
```

You can use `DialOptions` to set the auth credentials (e.g., TLS, GCE credentials, JWT credentials) in `grpc.Dial` if the need arises. Once the gRPC channel is setup, we need a client stub to perform RPCs. We get this using the `NewChordRPCClient` method provided in the pb package we generated from our `.proto`.

Now let's look at how we call our gRPC methods. Note that in gRPC-Go, RPCs operate in a blocking/synchronous mode, which means that the RPC call waits for the server to respond, and will either return a response or an error. Calling the simple RPC `GetFeature` is nearly as straightforward as calling a local method.

```
response, err := client.GetPredecessorIdCaller(context.Background(),
                                                    &pb.RemoteNodeMsg{})
```

5 GoBook

For our lab, we will create a small distributed social networking platform called "GoBook". GoBook consists of a central news feed that students can post to using their GoBook clients. Other students can read the news feed and "Like" posts as well as comment on them. Students can also send a PM (Private Message) to other students.

The wonder of using gRPC for GoBook is that every student can write the client in their own language provided a common .proto file is being used.

You are required to do the following:

- Create a GoBook Server. The GoBook server should register users and store relevant information such as IP address and username. The server will also have a news feed that users can post new posts to or comment/like existing ones. The server should also be able to provide a list of users currently online.To call gRPC methods, we first need to create a gRPC channel to communicate with the server. We create
- Create a GoBook Client. The client should register its user with the server. The client can also view the current news feed and provide functionality for posting new posts or commenting/liking existing ones. Clients can also send other users PM's by asking the server for their IP adresses.

Note that we are keeping the specification of GoBook vague, so you have control of the design. This is to give you practice for Puddlestore, where you will have full control over the design of your API calls.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS1380 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s18/feedback.html`.