Project 1[1]
CS 118: Cloud Computing

Assigned: 09/24/25
Due: 10/15/25 at 11:59 PM Medford Time
HW should be completed in your presentation group
Please submit on Gradescope in .zip format or via Github option

Contents

1 Introduction

This project is intended to get you more familiar with gRPC. The official docs are an excellent resource for learning gRPC concepts. You can learn how to set up and install everything via this quickstart guide.

gRPC is Google's RPC framework. An RPC framework allows you to execute functions remotely - a function pointer (and arguments for said function) get passed to another computer that executes it, and the result is sent back to your local computer. In practice, RPC frameworks look and behave very much like traditional HTTP requests, and, in fact, gRPC uses HTTP/2 under the hood.

[1] Last updated 09/24/2025 by Zhaoqi(Roy) Zhang

So why use gRPC instead of a vanilla HTTP request? Because (a) gRPC has higher throughput than regular HTTP, and (b) it lets you specify the schema for your server's entire API just once, and then automatically generate code implementing your API in multiple programming languages. It takes away some of the pain of updating distributed systems components - if you make changes to your server API, you can easily update clients by just having them import the newly generated code.

gRPC itself is built on Protocol Buffers. Think of protocol buffers as an alternative to JSON or gzip for formatting messages before they are sent over a network (the **wire format**). Protocol buffers convert objects in a programming language to a highly-compressed binary representation that can be sent over the network, allowing throughput to be very high.

A consequence of this design is that both client and server must implement custom converters (serializers and deserializers) to parse message formats back into objects for your programming language - the good news is that these converters are automatically generated for you. All your code has to do is call it.

In this project, you will implement the core functionality of a simple chat application that utilizes gRPC and protocol buffers. You shall learn how to write an API specification using protocol buffers, how to generate code stubs implementing this API automatically, how to create clients and servers that invoke gRPC methods, and how to use some extra goodies (like metadata and interceptors) provided by gRPC - all while getting a nice distributed service out of it too!

2 Get Started & Submission

Here is the link to our GitHub template repo that contains the starting code. This assignment will be done on a group basis. GitHub makes team collaboration easier.

Start & Submission note:

1. Don't clone our template repo. Use create repo with template instead. You have to work on the created repo.
    a. As the TA will read your code, you will have to push your code to your own repo to get grades.
2. On submission, upload the following stuff to Gradescope:
    a. Compressed .zip file of everything you have in the created project repo
        i. A text file containing the URL of your created repo for this project.
    b. Your project 1 GitHub repo that is linked to Gradescope

A brief high-level overview of the application we're building (called WhatsUp) follows.

We've kept the design deliberately simple for pedagogical purposes:

1. At heart, it resembles a very naive email service. Users use a client to connect to a server, can send messages to users currently logged-in with the server, and can check for new messages. No additional features - like push updates, folder organization, or message filters - are provided.

2. Our service is both stateful (users are either logged-in or not) and offers some security: you shouldn't be able to log in from a different client as a user if that user is logged in on a separate client. To implement this, we have a notion of *authentication*. When users connect, they are given a unique authentication token that they must provide in all future requests. When they disconnect, they must explicitly invalidate the token. Our tokens never expire - we expect our clients will never abruptly disconnect (an assumption you should never rely on in production!).

In the assignment, you will find three folders / files:

1. `client` and `pkg/client_core.go`: Contains code for a WhatsUp client. Its primary job is querying the server and rendering query results for the command line.

2. `server` and `pkg/server_core.go`: Contains code for a WhatsUp server. Its primary job is to maintain state to verify authentication, and store messages from users until a client retrieves them.
3. `pkg/whatsup.proto:` The protocol buffer definition. You will generate code from this file.

WhatsUp client allows you to send and fetch messages. It also allows you to list all logged in users. In this assignment, your task will be to implement sending and fetching messages, as well as logging in and ensuring the received authentication token is used in every subsequent message until logout.

3 Requirements

To receive full credit for this project, we will build and run your binary against integration tests we've written. If your code passes these tests on your working Linux machine, you should be fine!

For your benefit, the source code for these integration tests is included in this repository.

4 Specifying an API with Protocol Buffers

Let's take a look at a piece of our `whatsup/proto` file:

```
syntax = "proto3"; // required boilerplate - always have this at top
package whatsup; // used in the generated code, see example below

message Registration {
    string source_user = 1;
}

message AuthToken {
    string token = 1;
}

service WhatsUp {
    rpc Connect(Registration) returns (AuthToken);
}
```

This simplified `.proto` file essentially says: there exists an API endpoint (belonging to our service `WhatsUp`) called `Connect`. This endpoint accepts a message of type `Registration` and responds with a message of type `AuthToken`. This information is all indicated by the `rpc` keyword in the `service WhatsUp` object.

An `rpc` can accept *only* one input type and *only* one return type, and (if not otherwise specified) will close the endpoint connection once one message of each type is exchanged, exactly like a regular HTTP request.

A message itself is just an aggregate containing a set of typed *fields*. In the above example, `Registration` has a single string field called `source_user`, and `AuthToken` has a single string field called `token`. The `= 1`, `= 2` etc. markers on each of these fields are required syntactic sugar - under the hood, Protocol Buffers uses them to order these fields in the binary format of the message. Tags must start from 1.

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, mark some fields as `optional` and even use enums. A field may be repeated any number of times (including zero) if the `repeated` keyword is added before the type. The order of the repeated values will be preserved in the Protocol Buffers. You can 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 information should be more than sufficient for this class.

Note: technically, only messages are part of the protocol buffers specification. Service objects and the rpc keyword belong to gRPC, which extends the protocol buffers specification. While it is rare to use protocol buffers without also using gRPC, knowing this will help you greatly when looking up documentation - in general, annotations that seem network-specific (rpc, stream, service) will be found in the gRPC documentation; all others will be in the protocol buffers documentation. The reason for this split is because protocol buffers are meant to be use case-agnostic, while gRPC is not.

5 Implementing an API

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.

Download and install `protoc` if you haven't. Be sure to add `protoc` to the `PATH` variable for your environment, so that you can call `protoc` from the command line.

Assuming you are in the root of the assignment's directory, we can run the following commands:

```
# For the first time, get protoc-gen executable

# Locally
go get google.golang.org/protobuf/cmd/protoc-gen-go
go get google.golang.org/grpc/cmd/protoc-gen-go-grpc

# To generate go code from proto while in the root folder of this
project
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=.
--go-grpc_opt=paths=source_relative pkg/whatsup.proto
```

The above command will generate corresponding Go code for all proto files in the current working directory. **For convenience, a Makefile has been provided in this project that will run the last command for you.**

When you now inspect the whatsup folder, you should see two new files generated for you:

1. `pkg/whatsup.pb.go`: This file contains the generated code for all of the protocol buffer *messages*.

2. `pkg/whatsup_grpc.pb.go`: This file contains the generated *client* and *server* code for your RPCs

You don't have to inspect these files too thoroughly - much of it is deep implementation code needed for gRPC to work properly. There are a few key things you should take note of, though.

Message Types have Become Go Structs

In `whatsup.pb.go`, all of your message types have now been implemented as structs - for example, the protobuf type

```
message Registration {
    string source_user = 1;
}
```

has been converted into the Go struct definition

```
package whatsup // produced by the `package` keyword in .proto

…

type Registration struct {
      state         protoimpl.MessageState
      sizeCache     protoimpl.SizeCache
      unknownFields protoimpl.UnknownFields

      SourceUser string `protobuf:"..."`
}
```

This means you can now create Registration objects like any other Go object:

```go
import (
    "whatsup/whatsup",
    "fmt"
)

func main() {
    r := whatsup.Registration{SourceUser: "foo"}
    fmt.Println("%+v", r)
}
```

RPC Definitions Have Become Go Interfaces

In `whatsup_grpc.pb.go`, your RPC call has now been implemented as interfaces satisfied by a client and a server - for example, the RPC call

```
service WhatsUp {
    rpc Connect(Registration) returns (AuthToken);
}
```

has been converted into the following Go interfaces and helper functions / structs

```go
package whatsup // produced by the `package` keyword in .proto

type WhatsUpClient interface {
      Connect(ctx context.Context,
              in *Registration,
              opts ...grpc.CallOption) (*AuthToken, error)
}

// returns a struct that implements WhatsUpClient
func NewWhatsUpClient(_ grpc.ClientConnInterface) WhatsUpClient {...}

type WhatsUpServer interface {
      Connect(context.Context, *Registration) (*AuthToken, error)
}
```

```
// this becomes important when implementing WhatsUpServer interface
type UnimplementedWhatsUpServer struct {}
```

For clients, the interfaces have already been implemented by gRPC, so you can just ask for a new client using `whatsup.NewWhatsUpClient()`:

```
import (
    "whatsup/whatsup",
    "google.golang.org/grpc",
    "context",
    "fmt"
)

func main() {

      // Establish a connection to the chat server
      connection, _:= grpc.Dial(
            // specify the address of the server - example below
            "localhost:8000",
            // indicate we should connect using plain TCP without SSL
            grpc.WithInsecure(),
            // block thread until connection is established
            grpc.WithBlock(),
      )

      // create a new gRPC client over this connection
      client := whatsup.NewWhatsUpClient(connection)

      // send a message that returns an AuthToken; blocks until done
      auth, _:= client.Connect(
            // take a look at the `context` standard library
            context.Background(),
            // our payload
            &whatsup.Registration{
                SourceUser: user,
            }
      )
```

```
      // should print whatsup.AuthToken{Token: "..."}
      fmt.Println("%+v", auth)
}
```

Server code is slightly more complicated in two ways:

1. Before being used, it must be *registered* with a "true" server listening on a port.
2. gRPC does not let you use its default implementations for the server interface - you must "subclass" `UnimplementedWhatsUpServer` and implement the methods yourself.

```
import (
    "whatsup/whatsup",
    "google.golang.org/grpc",
    "context",
    "net",
    "fmt"
)

// a new type that implements whatsup.UnimplementedWhatsUpServer
type server struct {
     whatsup.UnimplementedWhatsUpServer
}

// an example implementation of our server interface
func (s server) Connect(_ context.Context, r *whatsup.Registration)
(*whatsup.AuthToken, error) {
    token := r.SourceUser + " has been authenticated"
    return &whatsup.AuthToken{Token: token}, nil
}

func main() {
     realServer := grpc.NewServer()
     whatsup.RegisterWhatsUpServer(realServer, server{})

     // example port and address
     listen := net.Listen("tcp4", "localhost:8000")
```

```
     if err := realServer.Serve(listen); err != nil {
           fmt.Printf("failed to serve: %v", err)
     }
}
```

6 Context and Metadata

Our generated client interface seems to ask for a `context.Context` object. What is this mysterious entity?

Think of `context.Context` objects as HTTP headers or cookies. They can store values that a request can supply as meaningful *metadata* - for example, a context might contain an authentication token that servers can inspect before honoring the request. Contexts enable an advanced distributed systems debugging technique known as *distributed tracing*, which allows you to collect performance data across all the servers and functions touched by a single request, by storing all the performance data seen so far inside a request.

Contexts have some special properties that HTTP headers or cookies don't. Contexts can be programmatically *cancelled*, letting the server know that the user has requested the connection to close prematurely. They can also have *deadlines*, killing the gRPC request if it does not complete within a certain amount of time. These properties make contexts very powerful, and can be used in many systems as a way to coordinate work concurrently or at scale. A good overview of contexts is available in the official docs.

In our WhatsUp application, we store authentication tokens retrieved by `Connect` inside a *single* context. This context is then passed on to all other requests within the users' session. Contexts *erase* the type of the keys and values stored in them, converting them to `interface{}` objects, so we recommend gRPC's official solution to this: the metadata package, which ensures all values are stored concretely as strings (keys) and a list of strings (values).

```
import (
    "context",
    "google.golang.org/grpc/metadata",
    "fmt"
)
```

```go
func main() {
    ctx := context.Background()
    // store a key-value pair inside a context
    ctx = metadata.AppendToOutgoingContext(ctx, "key", "value")

    // extract the key-value pair before being sent
    md, _:= metadata.FromOutgoingContext(ctx)
    // Note: to read the same data on the server, use
    // metadata.FromIncomingContext

    // prints a slice, not a string - []string{"value"}
    fmt.Println("%+v", md["key"])
}
```

7 gRPC Interceptors

gRPC interceptors essentially serve as middleware for gRPC calls. Client interceptors capture the request before the client sends it off to the server. Server interceptors receive the request before the server processes it. Some use cases for interceptors could be setting default timeouts, authentication, logging, and testing. See here for a more in-depth explanation of interceptors.

In our WhatsUp application, we have implemented and registered a server interceptor for you that checks if the accompanying request has the appropriate authorization token. This interceptor then inserts the actual username into the request's context before calling the method it was originally supposed to call. It's not important to know too much about them, so we won't touch too much on them - just that they exist, and serve an important role in building many gRPC applications.

8 Your task

You are expected to complete the core functionality of the WhatsUp application by implementing two new RPCs (Send and Fetch) using custom message types. You are also expected to fill out `Register`, a client function that calls our Connect RPC and returns a context object populated with the authorization token.

- Complete the demo.proto file that defines these two RPC services
- Generate `.pb.go` file from `demo.proto`
- Implement the RPCs on the server in the `server_core.go` file.
- Implement calling the RPCs on the client in the `client_core.go` file.

9 Handin

Once you finish the project, you should submit either

- A compressed .zip file of everything you have in the created project repo to [Gradescope](#).
    - Make sure all files are in the root directory of the submitted .zip file
    - Also, upload a text file containing URL to your GitHub repository along with the code.
- Or Github repository directly to Gradescope (check [this guide](#) for details)

The autograding will run and give you feedback. You can resubmit the project multiple times.

Acknowledgements

Special thanks to Theo Benson at Brown University (at CMU now), as we reused some materials from his distributed system course.