Using gRPC with Go

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

In the past few articles, we've used Go's `net/rpc` library to build a simple file metadata server. In this article, we're going to look at using gRPC (https://grpc.io) to fulfill the same purpose.

gRPC has many advantages over the built-in RPC library, namely:

- Fast and efficient network communication
- Ability to stream inputs and outputs
- Automatic transport encryption
- Ability to interact with other languages
- Ready extensions to support authentication and connection handling.

gRPC is typically boiled down to the description "Protobuf messages over HTTP/2." This is true to a first pass, but it also encompasses the libraries, middleware extensions, and interactions with other languages.

For the sake of brevity, some sections of the code examples here are left out. The full code for this example can be found at https://github.com/cmceniry/login-grpcls.

Getting Started

Our service building story goes a little something like this:

- First, we have to get the `protobuf` tools and gRPC Golang libraries.
- Next, we'll create a language-independent `protobuf` definition for our service.
- With the `protobuf` definition in hand, we'll generate Golang hooks.
- After that, we can fill in our interactions with those hooks.
- And finally, we can compile and run.

Getting the Tools

Building gRPC applications requires a couple of tools. The first is `protoc` which is the language-independent protobuf compiler. The second is the `protoc-gen-go` plugin which is used to generate Golang code for the data and interface types. In addition, we're going to need the Golang `grpc` library.

The installation for `protoc` depends on your platform. It is packaged up for various platforms (rpm, brew, etc.), but when in doubt you can get it from the official release location: https://github.com/google/protobuf/releases.

For the Golang-specific items, you can grab these with the `go get` command.

```
go get -u google.golang.org/grpc
go get -u github.com/golang/protobuf/protoc-gen-go
```

The former is a library, and it is possible to alternatively vendor it and manage it with dep. The latter produces the protoc-gen-go executable, so it needs to be installed in such a way that that is available, typically with a go get into the home workspace. However you choose to install it, you will want to make sure that its location is in your PATH since protoc will look there for it. For the above, you can use

```
export PATH=${PATH}:~/go/bin
```

Now that we have a good environment, we can move on to working on the code. If you're following the go get commands from above, you can grab the example code and change into that directory now. This is dependent on the TLS credentials generated by the previous login-glss, so we'll need to get that and create the certificates first.

```
go get -u github.com/cmceniry/login-glss
go get -u github.com/cmceniry/login-grpcls
cd ~/go/src/github.com/cmceniry/login-glss
go run certs/generate_certs.go
cd ../login-grpcls
```

The Protobuf Definition File

Like any protobuf protocol, gRPC starts with a .proto file. On a first pass, this is a simplified description of the messages that will be transported over the connection. Specifically for gRPC (well, RPCs in general, but other implementations are rare), it is also a description of the services and their interfaces, which use these messages.

To determine what should be in our .proto file, we need to think about what we're passing back and forth between the client and the server. To mirror glss, we will want to pass from the client to the server the Path that we're going to be using. From the server to the client, the resulting File information blocks for the client-supplied path. In addition, we want an RPC to call LS. The RPC semantics in .proto also define a service, Lister, which encapsulates several RPCs and properties.

So we'll want to define the following four items in our .proto: Path, File, LS, and Lister, for which we'll need a bit of boilerplate. We are telling protobuf which version we'll be using, and we need to wrap our collection of services and messages into a package.

```
syntax = "proto3";
package directorycontents;
```

Next, we need to define what will be transporting over our connections: Path and File. We will structure these as message items that will be used in our remote calls. message is the generic type for data passing between the client and server, regardless of whether it is a parameter or return value. Each message is a combination of a message type name and specific typed fields that are of meaning in that message.

First, we'll tackle Path, which just has a single field in it, string name.

```
message Path {
  string name = 1;
}
```

The 1 associated with name is a field numeric ID to allow for compatibility between clients and servers of different versions. This allows for nonbreaking changes to the API without having to upgrade *every* client and server out there. You can enable new fields by appending to the end with a new number. You can change existing fields by adding a new field with the appropriate changes for the old field. You will end up populating both fields for a period of time, but it does allow for newer servers and clients to speak to both current and old versions of themselves.

Next, we'll build out our File response. It also has a string name, as well as size, mode, and modtime:

```
message File {
  string name = 1;
  int64 size = 2;
  string mode = 3;
  string modtime = 4;
}
```

Now that we have our two message definitions, we can move on to our service definition. Since we're providing a generalized directory lister service, we're going to call this Lister. As mentioned, this will contain our collection of RPC calls (in this case, it's just one). Each RPC has a list of inputs and outputs (in this case, it's just Path and File).

```
service Lister {
  rpc LS (Path) returns (stream File) {}
}
```

As mentioned in the introduction, gRPC allows us to stream inputs and outputs. In this case, we're going to call LS with a single input item Path, but we're going to get back a large list of File blocks. For efficiency and demonstrative purposes, we're going to stream the File responses—hence the stream modifier in the LS return values. We could have wrapped them all up in an array, but this way we don't need to maintain all of that in memory as we go through it. As we'll see in the application code, once a file is found, it can immediately be sent back to the client.

Generating gRPC Code

Now that we have the data types and function-call semantics, we want to put this language-independent form into something that

we can use in Golang. This is where `protoc` and `protoc-gen-go` come into play. From the root of our project:

```
protoc --go_out=plugins=grpc:. \
        directorycontents/directorycontents.proto
```

This invokes `protoc` and tells it to use the `proto-gen-go` with the gRPC plugin to process our `.proto` file. This will produce `directorycontents/directorycontents.pb.go`.

The full file is a bit much to go over in this article, but its key contributions are:

- It defines Go native structs for `Path` and `File`:

```
type Path struct {
        Name string protobuf:"bytes,1,opt,name=name"
json:"name,omitempty"
}
...
type File struct {
        Name    string protobuf:"bytes,1,opt,name=name"
json:"name,omitempty"
        Size    int64  protobuf:"varint,2,opt,name=size"
json:"size,omitempty"
        Mode    string protobuf:"bytes,3,opt,name=mode"
json:"mode,omitempty"
        Modtime string protobuf:"bytes,4,opt,name=
        modtime"
json:"modtime,omitempty"
}
```

- These each have some accessor functions to them, or you can manipulate the struct field directly.
- It defines a `ListerClient` interface (with accompanying constructor func). This interface is how we're going to call the gRPC functions from our code. Specifically, we're going to be calling the `LS` function on the return `ListerClient` interface.

```
type ListerClient interface {
        LS(ctx context.Context, in *Path, opts
...grpc.CallOption) (Lister_LSClient, error)
}
...
func NewListerClient(cc *grpc.ClientConn) ListerClient {
```

- It defines the `ListerServer` interface for the server side. We're going to build a struct that implements this interface as our way of responding to gRPC calls. Related to this, it defines the `Lister_LSServer` interface that is used specifically for our outlet to send responses to the `LS` calls.

```
type ListerServer interface {
        LS(*Path, Lister_LSServer) error
}
...
type Lister_LSServer interface {
        Send(*File) error
...
```

The Server Implementation

Now it's time to focus on our application code, starting with the server side. gRPC has presented us with an interface that we need to implement. As mentioned in the last section, we're going to implement the `ListerServer`.

Since this is Golang code, let's take care of the imports. Common practice is to import the generated `.proto` definitions with the alias name of `pb`. In addition, we're going to import the `grpc` library itself, and the `grpc/reflection` library to support API information sharing.

```
import (
    pb "github.com/cmceniry/login-grpcls/directorycontents"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/reflection"
```

Next, we'll divide the server into two parts. The first is the actual remote procedure to be called `LS`. The second is wiring up everything.

Since `ListerServer` is an interface, we need to set up two parts to it: a struct and the supporting member methods.

```
type server struct{}

func (s *server) LS(p *pb.Path, fileInfoStream
pb.Lister_LSServer) error {
```

You can wrap this up with much more, but for this example, our struct is as simple as can be. The `func` signature for `LS` must match the one from `directorycontents.pb.go`. Note that the gRPC response values are not a part of the return values from the function. Since we're going to stream the results back, we will be working with the `fileInfoStream` value as our conduit to send the data back while inside of our function.

The remainder of the func uses `filepath.Walk` just as the original gls and glss servers did. It only has modifications to handle sending the data directly back on the `fileInfoStream`.

```
        err := filepath.Walk(p.Name, func(path string, info
os.FileInfo, err error) error {
```

```
f := &pb.File{
    Name:    info.Name(),
    Size:    info.Size(),
    Mode:    info.Mode().String(),
    Modtime: info.ModTime().Format("Jan _2 15:04"),
}
err = fileInfoStream.Send(f)
```

In this, we're converting every `os.FileInfo` we see as we receive it. For it to go over the gRPC connection, it has to be in the form of the messages from the `.proto`. Here, we convert it to `pb.File`, which allows us to send it back using `fileInfoStream.Send`. To reiterate, this is happening as we see every file, so we don't have to construct any intermediate arrays before sending them all back.

Finally, on the server, we need to wire the network level up to our `ListerServer`. We'll start with a standard TCP network listener.

```
func main() {
    l, err := net.Listen("tcp", ":4270")
```

Since we want to enable TLS authentication, we need to prepare that. The first part is to load in the certificates and keys. This is identical to the way that we loaded them in `login-glss`.

```
certificate, err := tls.LoadX509KeyPair(
    "../login-glss/certs/server.crt",
    "../login-glss/certs/server.key",
)
if err != nil {
    log.Fatalf("could not load client key pair: %s", err)
}
caCert, err := ioutil.ReadFile("../login-glss/certs/CA.crt")
if err != nil {
    log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
```

With the credentials loaded, we need to format these for gRPC to use. This involves wrapping a typical `tls.Config` struct with a `grpc.credentials` struct. It's the latter that is used by the gRPC services. Much like the glss server, we need to provide our certificate and configure the pool and flag for client auth. In addition, we need to ensure that our expected TLS ServerName is supplied so that the client can validate against that.

```
creds := credentials.NewTLS(&tls.Config{
    ServerName:   "localhost",
    Certificates: []tls.Certificate{certificate},
    ClientCAs:    caCertPool,
    ClientAuth:   tls.RequireAndVerifyClientCert,
})
```

Next, we create a `grpc.Server` struct, then register a ListerServer with it. When creating the `grpc.Server`, we indicate that we're using the TLS configuration that we just set up.

```
s := grpc.NewServer(grpc.Creds(creds))
pb.RegisterListerServer(s, &server{})
```

Next, we enable information on the service via the `grpc/reflection` library. This is an optional step that allows generic gRPC clients to interact with our `Lister` service. You can inspect the information exposed using the gRPC command line tool found at https://github.com/grpc/grpc/blob/master/doc/command_line_tool.md.

```
reflection.Register(s)
```

Finally, we can start the `grpc.Server` by telling it to act on our `tcp.Listener`.

```
err = s.Serve(l)
```

If all has gone well, we've successfully wired our server together. Now, on to the client side. Other than the import, it consists strictly of a `main` func. Like glss, it takes a single command line argument, which is the directory to get the listing.

The Client Implementation

The new imports for the client all involve the gRPC libraries. The client refers to the same `.proto`-generated definitions as the server, so it will need to import them as well. And, obviously, it needs to import the `grpc` library.

```
import (
    pb "github.com/cmceniry/login-grpcls/directorycontents"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
```

As the first part of our main function, we need to load our TLS values.

Again, this is identical to how it is configured in glss.

```
certificate, err := tls.LoadX509KeyPair(
    "../login-glss/certs/client.crt",
    "../login-glss/certs/client.key",
)
if err != nil {
    log.Fatalf("could not load client key pair: %s", err)
}
caCert, err := ioutil.ReadFile("../login-glss/certs/CA.crt")
if err != nil {
```

```
        log.Fatal(err)
    }
    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
```

As with the server side, we need to wrap these as `grpc.credentials`. Note that our TLS config only requires that we supply our certificate and provide a RootCA pool against which to validate the server.

```
    creds := credentials.NewTLS(&tls.Config{
        Certificates: []tls.Certificate{certificate},
        RootCAs:      caCertPool,
    })
```

Next, we form our network connection. Unlike on the server side, the `grpc` library has its own Dialer for the client side. We need to supply this with the endpoint to connect to and our general gRPC configuration—in this case, our credentials configuration.

```
    conn, err := grpc.Dial("localhost:4270",
grpc.WithTransportCredentials(creds))
```

At this point, we've only established a general gRPC connection. There is nothing specific about our particular API, so we need to remedy that. We accomplish this by wrapping the general gRPC connection with a client that is specific to our Lister service.

```
    c := pb.NewListerClient(conn)
```

Now we can make our RPC call. This uses the context library for handling timeouts and cancellations. In this example, we're just going to use the `context.Background()`, so we're skipping over additional `context` library handling of timeouts and cancellations. Our actual argument to `LS` is the `pb.Path` wrapped value from the command line.

```
    files, err := c.LS(context.Background(), &pb.Path{Name:
os.Args[1]})
```

Since the return value from `LS` is a `protobuf` stream, we need to read each value from it. We do this by looping around `files.Recv()`. If the stream is complete, `LS` returns the `io.EOF` sentry error and allows us to break out of the loop. Otherwise, unless there's an error, we print out of the file information.

```
    for {
        f, err := files.Recv()
        if err == io.EOF {
            break
        }
        if err != nil {
            log.Fatalf("LS file failure: %s", err)
        }
            fmt.Printf("%s %10d %s %s\n", f.Mode, f.Size,
f.Modtime, f.Name)
    }
```

Running It All

Now that we have the client and server, we can run it all together.

```
login-grpcls$ go run server/main.go &
[1] 11488
login-grpcls$ 2017/12/16 21:44:17 Starting server
login-grpcls$ go run client/main.go .
drwxr-xr-x       204 Dec  6 21:27 .
drwxr-xr-x       102 Dec 16 11:43 client
drwxr-xr-x       136 Dec 13 19:16 directorycontents
-rw-r--r--       267 Dec  6 21:27 links
drwxr-xr-x       102 Dec 16 11:32 server
```

Conclusion

This article has provided a brief introduction to using gRPC with Golang. In addition, this series of articles has given us two implementations for our `LS` service—one using the `net/rpc` from Golang, and one using gRPC. I hope that you now feel comfortable enough to consider using gRPC in your work and, more importantly, to be able to weigh the pros and cons of when to use it or `net/rpc` as appropriate for your situation.

A few specific similarities and differences to remember between the two:

- Both setups involve configuring a generalized RPC server and then registering calls to it.
- Outside of some wrapping, both interact with TLS in the same way. The underlying implementation at the TLS layer is the same. Given the end-to-end principle, it should not be surprising to see the same behavior from a wrapping layer.
- With `net/rpc`, we're handling Golang data structures. With `grpc`, we're handling more generic data structures (which can be referenced by multiple languages). The `net/rpc` way is easier to handle in Golang but does limit the interaction to Golang. Which one you should use depends on the users of your API and the contract you need or want to maintain.
- While we did not demonstrate it in this example, gRPC has several middleware wrappers. These provide higher-level API enrichments to help enable resiliency and visibility. Since there are interface patterns, there is the possibility that the same exists for `net/rpc`, but its goal has been to be a solid simple standard library. It's unlikely that these will exist for `net/rpc`.

Happy Going!