**Exercise 12**

*gRPC / Binary protocols*

**Pre-reqs**

None

**Objectives**

Demonstrate the use of a fast binary protocol

Demonstrate a python client calling a typescript service via ProtoBuf messages

Use server-side streaming

**Steps**

1. Firstly, we need to checkout the code from Github.   
   In a new terminal window, type the following:

cd ~  
git clone https://github.com/pzfreo/grpc-sample-ts.git  
cd grpc-sample-ts  
yarn install

1. If you look at the code, you will see that there are some missing definitions (e.g. these errors:)  
   
2. That is because we need to generate code from a protobuf definition using some scripts.   
     
   Let’s first look at purchase.proto  
   
3. This contains the gRPC service definition in the ProtoBuf format.  
     
   You can see there is a simple input message and output message. The information may be similar to the REST model, but the approach is definitely different. There are no resources, no well defined error/return codes, and no standard VERBS. Certainly no HATEOAS!
4. Let’s generate the code from the proto. There is a yarn script “generate” that calls a script in ./scripts to generate the code using the typescript grpc tools. These are installed by the yarn install command (as they are defined in package.json).  
     
   yarn generate



1. Look at the code. There are two key files:  
   

As you can see this defines a function “purchase” which creates a reply. You can see the code just prints out the input message and returns a new UUID and rc=1.

1. And then another program file sets up the listener:  
   
2. You can run this code using:  
     
   yarn start
3. Before we create a client, let’s try a generic client. There are a couple of useful ones.  
     
   One is called grpcurl which is clearly meant to be like cURL.  
     
   We can install it either by downloading the binary or using go:

go get github.com/fullstorydev/grpcurl/...

go install github.com/fullstorydev/grpcurl/cmd/grpcurl

1. In the directory with your grpc code in, try:  
     
   grpcurl -proto proto/purchase.proto list

freo.me.purchase.Purchase

1. Now try   
   grpcurl -proto proto/purchase.proto describe

freo.me.purchase.Purchase is a service:

service Purchase {

rpc purchase ( .freo.me.purchase.PurchaseRequest ) returns ( .freo.me.purchase.PurchaseReply );

}

1. This is quite hard work though. Wouldn’t it be nice to have a graphical grpc client?  
     
   cd ~/Downloads

wget <http://freo.me/bloomrpc-deb> -O bloomrpc.deb  
sudo dpkg -i bloomrpc.deb

bloomrpc  
  


1. Add the proto file, and set the server target to be localhost:50051



Click on the purchase method and it will autofill a sample:  


Click play:



You will see how long it takes as well (14ms in my case)

Now let’s create a Python client. In a new terminal window:  
  
cd ~/grpc-sample-ts  
mkdir python  
cd python  
cp ../proto/purchase.proto .

1. We need to install the python grpc tools:

sudo pip install grpcio  
sudo pip install grpcio-tools

sudo pip install --upgrade protobuf

1. Now we can run the client code compiler (all on one line please)  
     
   python -m grpc\_tools.protoc -I . --python\_out=. --grpc\_python\_out=. ./purchase.proto
2. If you look at the directory (ls), you will see two new files:  
   > ls -l

total 20

-rw-rw-r-- 1 oxsoa oxsoa 1433 Jan 10 19:24 purchase\_pb2\_grpc.py

-rw-rw-r-- 1 oxsoa oxsoa 8269 Jan 10 19:24 purchase\_pb2.py

-rw-rw-r-- 1 oxsoa oxsoa 446 Jan 10 19:21 purchase.proto

1. These files will let us call the purchase service. Create and edit a new python program:  
     
   code purchase-client.py
2. Type in the following code: (available here: <http://freo.me/grpc-py> )
3. Run the program:  
   

python purchase-client.py

1. Check the console log on your Typescript server and you should see it has been called. Congratulations - we have successfully built a heterogeneous gRPC app.
2. One of the key features of gRPC is the support for client, server and bidirectional streaming. This is really useful, especially for getting push notifications from servers without needing clients to be internet-addressable. Let’s try this out.  
     
   First, let’s add the following protobuf message definitions to the grpc-sample-ts/purchase.proto files:  
     
   message Empty {}

message Update {

string msg = 1;

}

1. Also add the following rpc definition:  
     
   rpc push (Empty) returns (stream Update) {}
2. Regenerate the server code:

yarn generate

1. You will see that the types no longer match in the PurchaseServer.ts  
   We need to add a push function:  
     
   This code will write the time back as a message 10 times, with a 1 second (async) pause between calls. This is simulating a system that occasionally sends updates
2. You will need to fix the imports too.
3. Restart the typescript server.
4. Go back to Bloomrpc and reload the proto file.
5. Try out the push method:



Notice the extra responses coming back one second apart:  


1. Copy the proto to the python directory and regenerate the python bindings.
2. Add this code to your purchase-client.py (and comment out the other call)  
     
   for update in stub.push(purchase\_pb2.Empty()):

print (update.msg)

This is really rather elegant code!

1. You should see streaming working:  
   
2. Congratulations - you have completed the lab!

**Extensions**

1. Change the python code to call the server 10000 times. Add a timer around the code: e.g.  
     
   import time  
     
   t0 = time.time()  
   for x in range(0,1000):  
    //code here

t1 = time.time()  
t = t1-t0  
print("elapsed seconds: " + str(t))

1. See how long it takes. Note that if you run it more than once the node server may “warm up”, so the numbers should improve and then stabilize.
2. You could also remove all the console logging in the Typescript and Python code and try again. What is the throughput of requests once the system is warm?
3. How does this compare to our throughput with wrk?  
   Is this a fair comparison?