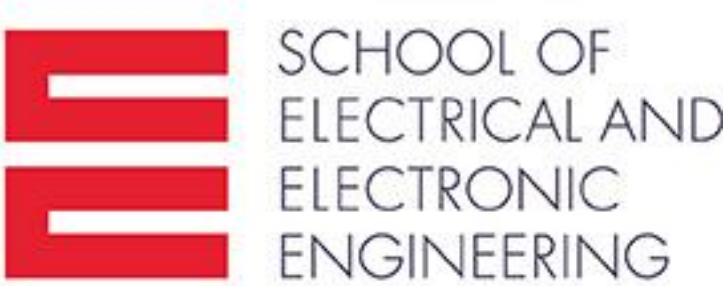

Network Centric Computing

Formal Element: gRPC Calculator

by

Jack Harding

This Report is submitted in partial fulfilment of the requirements of the Honours Degree in Electrical and Electronic Engineering (DT021A) of the Dublin Institute of Technology

20 August 2019

Lecturer: Dr Ray Lynch

Contents

Introduction

gRPC is an open source, multi-platform framework enabling remote communication between client and server and is known for its low latency, highly scalable nature. Like RPC, the services are defined in a definition file where the input and output parameters are defined, the same is done with a proto file; a message is declared which describes the contents of the message e.g. a phone message might have a string value for the model and a integer value for the price. [1]

When compared to traditional RPC, gRPC allows for far more efficient communication and a range of available platforms. The improvement in efficiency is helped using HTTP/2 rather than 1.1:

- Multiplexing of TCP rather than queued
- Compresses headers reducing the required data redundancy
- Server push; instead of waiting for client requests the server anticipates the needed resources [2]

The introduction of the Protocol Buffer language greatly improved object serialisation and compatibility making it an appealing option for mobile applications, allowing for one service definition to be written and it to be used on multiple platforms, one channel can carry multiple TCP connections, increasing efficiency and battery life on those mobile devices. An example of a Protobuf request message below describes the request serialisation, which basically means when the client sends a request this is the format, they wish to parse the data with.

```
message Phone {
    string model = 1;
    int32 price = 2;
}
```

The integers values above do not indicate the value of the variable rather the size in bytes they occupy when serialised, each variable in a message must have a unique number attached to it, the range is 1-15 but can be increased. The similar approach is used on the response message, but the number/type of fields returned may differ, just like a normal function. Like JSON and XML but is more efficient due to its encoding,

making it far faster, when XML and JSON are compared to proto, the compressed data size is a third of XML and half that of JSON. [3]

Using the message from before as an example, the price will be serialised using base 128 varints, which is a way of serialising integers using one or more bytes. Each byte, except the last, has the MSB set, this tells the protobuf encoder that there are more incoming bytes, like a stop bit. To send the value of the *Phone* message, the remaining 7 bits store the two's complement of the value, with the lower value first [encoding]. So, if the price of the phone €650 or `10 1000 1010,` the two bytes are separated:

```
0000 0010 1000 1010
```

The MSB is then moved to the front (second byte), and another byte is added to store the MSB:

```
1000 0000 1000 1010 0000 0010
```

When encoding in proto, a message is a series of key-value pairs, the key represents two pieces of information: the field number (how frequently is it used) and the wire type (value type). For the price variable, it being a varint, uses 0 as its wire type and its field number was set to 2. The last three bits store the wire type, the rest are the field (typically 0-15), the key in binary format is below:

```
0001 0000
```

The price's 32-bit integer type, field number of two, and value of 650 is encoded to:

`0001 0000 1000 0000 1000 1010 0000 0010` or `10 80 8A 2`

Creating a service to use these messages, involves the use of both response and request messages:

```
service NewService {
    rpc NewFunc(NewRequest) returns (NewResponse);
    rpc AnotherFunc(AnotherRequest) returns (AnotherResponse);
}
```

NewFunc is the RPC function, which is passed NewRequest, this service returns the response message. The code above is written in the .proto file which is then compiled into client and server stubs. [4]

Objective

The objective of this report is to implement a client/server application based on the gRPC framework which can then be called remotely by a user application. The user application is based upon a Windows Forms calculator taking user input via GUI and passing it to the client.

Code

The first step in gRPC is to define a service in the .proto file, this service will need to be accompanied with request/reply messages as well as an RPC method to specify input/output types. There are different options for streaming, a single client request could be sent to open a server stream, vice versa where the client streams, and bidirectional too. This implementation only needs simple RPC as there's only need for a single client request and a server response when using a calculator.

```
package com.example.grpc;

message SquareRequest {
  float inval = 1;
}

message SquareReply {
  float outval = 1;
}

service SquareService {
  rpc RetSquare (SquareRequest) returns (SquareReply) {}
}
```

To generate the server and client interface, the proto file must be compiled using protoc. The proto code above is added to project in VS 2017, the NuGet dependencies are added (Grpc , Grpc.Tools, ProtoBuf) and the code below is added to the .csproj file.

```
  <Protobuf Include="square.proto" />
</ItemGroup>
```

The default Class1.cs is deleted and the project is built. Two files are generated and found in the */obj/Debug* directory, the first is *Square.cs* which has all necessary protocol buffer code to satisfy the reply and request types, second is the *SquareGrpc.cs*, this file acts like a stub for both client and server, including the client and base classes which allow. When this project is built its output is a DLL which is used by both client and server.

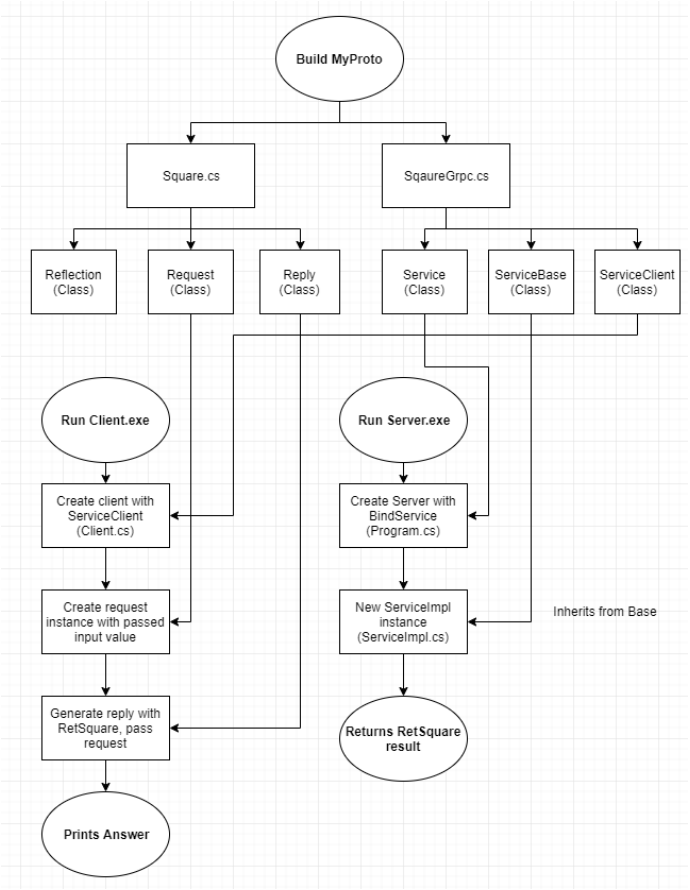

Figure 1 gRPC exe Flowchart

Figure 1 shows the flowchart for the console application version of the application. The reflection class is not used directly here but provides information about publicly available services that gRPC offers for both server and client at runtime to generate the necessary requests and replies. [reflect]

Server

A console application project is added to the MyProto solution above using .NET 4.5, all previous NuGet packages are added along with a reference to the *MyProto.dll.* The package made in the proto file is the namespace referenced in both the server and client, this gives access to the necessary methods to create the server.

```
using Com.Example.Grpc;
public class ServerProgram
{
    const string Host = "localhost";
    const int Port = 50051;
```

The port and IP are specified above and used below in the server configuration.

```
var server = new Server {
    Services = {
        SquareService.BindService(new SquaredServiceImpl())
    },
    Ports = {
        new ServerPort(Host, Port, ServerCredentials.Insecure)
    }
};
```

In *ServerProgram.cs* above, the *BindService* method is used to link the server to the implementation file *SquaredServiceImpl* (similar to binding handle in RPC), the reply and request methods declared in the stub file and added as a method to the *ServerServiceDefinition* using *AddMethod* which takes the proto method and the CS method to create an immutable (structure cannot change) server definition. The new *ServerPort* object takes the previously declared host and port numbers (endpoint) and adds security.

```
public class SquaredServiceImpl : SquareService.SquareServiceBase
{
    public override Task<SquareReply> RetSquare(
        SquareRequest request,
        ServerCallContext context
        ) {
        return Task.FromResult(
            new SquareReply {
            Outval = request.Inval * request.Inval
        });
    }
}
```

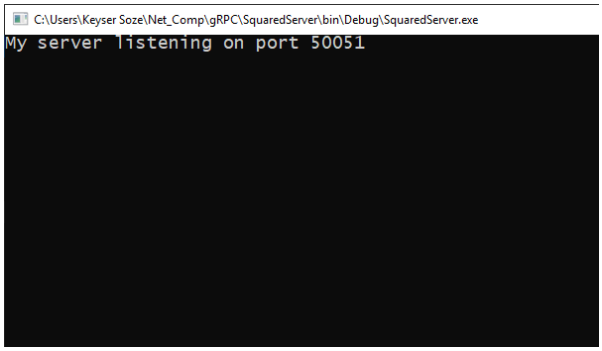

Figure 2 Server Running

The sever implementation inherits from the abstract *SquareServiceBase* class, just like the proto file, this class uses both request and reply messages to construct the *SquareService*, the request to the server is taken as a parameter to be used, then creates and returns the reply. This is to be built as a console application and ran in command line, the result is shown in Figure 2, this is left running awaiting requests from the client which it sends its result of *SquareReply* to.

Client

The client makes requests to the server, passing the required parameters and receiving the result. The client is developed similarly to the server starting with a .NET console application, reference to *MyProto.dll*, and all relevant gRPC packages. To begin, the client is built as an executable to test both client and server programs. The endpoint must be identical to that of the client.

```
var channel = new Channel(HOST + ":" + PORT, ChannelCredentials.Insecure);
```

Inside the main a new channel is created. The class channel in gRPC is made to define a connection to a remote server, this can be used by multiple client objects and is recommended due to it being an expensive operation. The constructor takes the endpoint as a string parameter and the security is specified using *ChannelCredentials*.

```
var client = new SquareService.SquareServiceClient(channel);
```

Accessing the *SquareServiceClient* method must be accessed by passing a parameter as it throws an exception as per the default constructor, the *ClientBase* class is used for client-side stubs and accesses the proto-generated *SquareGrpc.cs* file.

```
var request = new SquareRequest { Inval = 74 };
```

The input float value created in the proto file before is made equal to 74 for testing purposes and then passed to a *SquareRequest* class (message in proto), this inherits from Protobuf's *IMessage* class which facilitates serialisation.

```
var response = client.RetSquare(request);
```

The request is defined and can now be used by the *RetSquare* method in the *SquareServiceClient*, this method returns the call to the server implementation script which returns:

```
Outval = request.Inval * request.Inval
```

The channel is then shut down and the answer is printed.

```
channel.ShutdownAsync().Wait();
Console.WriteLine("Answer:" + response.Outval);
```

The client project is built as an executable and the server.exe is ran followed by the client's, the result is shown in Figure 3.

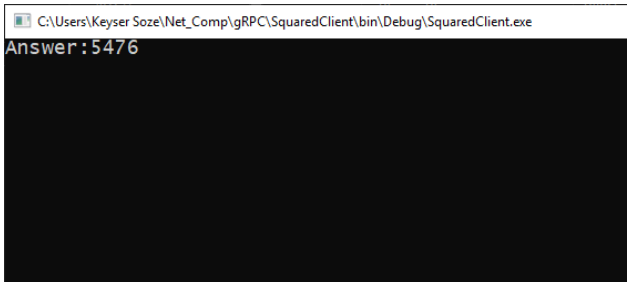

Figure 3 Reply from Server (Client.exe)

Calculator

This section uses the methods made previously in a user input Windows Forms application. The calculator previously created can be used with the gRPC client and server programs to take user input from a GUI and pass it to backend to return the answer. All NuGet packages must be added to this project also.

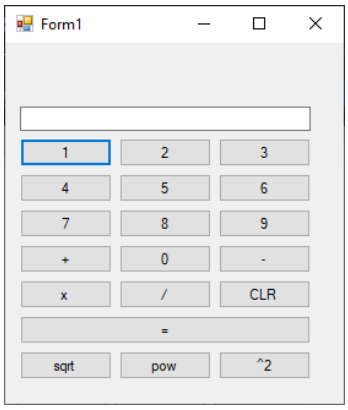

Figure 4 Calculator Interface

The following interface was used with the square button (^2) used to call the gRPC client, this button is linked to a C# script which provides an action upon pressing the button.

Before any buttons can be pressed, the client's DLL must be added as a reference, so the calculator has access to *RetSquare*. The following lines must be changed to convert the console application to a library. The main method is replaced by a float one:

```
public float RetSquare(float x) {
```

The console write lines are unnecessary now, the *Inval* is made equal to the input method's parameter (x), and a return line is added accessing the output value stored in the response variable.

```
var request = new SquareRequest { Inval = x };

return response.Outval;
```

The *SquaredClient* project is converted to a class library and rebuilt, the DLL created is inside the \bin\Debug directory of the project. This can now be added to the calculator project. The namespace of the DLL added as reference is then included as an assembly in the script.

```
using SquaredClient;
```

The value given by the button press is saved in the current value variable and used later. A new client program instance must also be created.

```
float curVal = Convert.ToSingle(textBox1.Text);
ClientProgram mySqaure = new ClientProgram();
```

The mySquare object is then used to return the square of the input value with RetSquare, this is then added to the running total (result of previous operations) and the text box is updated with the answer.

```
float sqrVal = mySqaure.RetSquare(runningTotal);

textBox1.Text = Convert.ToString(sqrVal);
runningTotal = sqrVal;
```

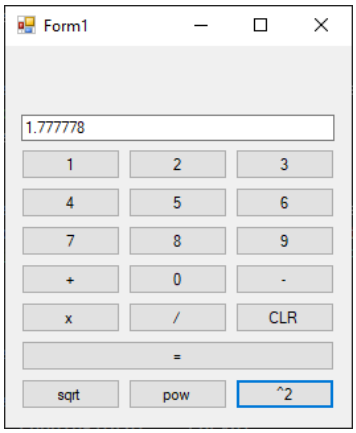

Figure 5 Calculator Tested

To test, the server's executable is ran in the command line as before and the calculator app is launched and a value is added to the text box.

4/3 was passed and the resulting float was returned.

References

[1] "Guides – gRPC", *Grpc.io*, 2019. [Online]. Available: https://grpc.io/docs/guides/. [Accessed: 19- Apr- 2019].

[2] "HTTP/2 vs HTTP/1 - performance comparison?", *ImageKit.io*, 2019. [Online]. Available: https://imagekit.io/blog/http2-vs-http1-performance/. [Accessed: 01- May- 2019].

[3] "5 Reasons to Use Protocol Buffers Instead of JSON For Your Next Service", *Codeclimate.com*, 2019. [Online]. Available: https://codeclimate.com/blog/choose-protocol-buffers/. [Accessed: 22- Apr- 2019].

[4] "Encoding | Protocol Buffers | Google Developers", *Google Developers*, 2019. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding. [Accessed: 25- Apr- 2019].