

*Title:* **Network Centric and Distributed Computing Formal Assignment: gRPC**

*Submitted by:* **Colm Doyle**

*Student Number:* **C15387731**

*Date Submitted:* **29 April 2019**

*Submitted for*

*Module:* **COMP4600 Network Centric and Distributed Computing**

*Programme :* **DT021A/4**

*Lecturers:* **Colm Doyle**

Contents

# Introduction

You are required to write a program and a report based on interfacing to Google’s version of RPC called gRPC, which is a language-neutral, platform-neutral remote procedure call (RPC) framework and toolset. It has been specifically designed for building scalable distributed client/server-model-based applications and cloud-based applications.

## Objective

The objective of this assignment is to study gRPC and write a programme which makes use of it. It is required that;

1. A study is performed on the background of gRPC and a comparison is done between gRPC and RPC.
2. A programme is written for a gRPC client and server, and then a button is added to the C# calculator that, when pressed, calls the gRPC remote function created and displays the return value on the calculator screen.

# gRPC

gRPC is an open source remote procedure call (RPC) frame initially developed at Google for object serialisation. Bi-directional streaming with HTTP/2 based transport is used, while Protocol Buffers are used as the interface description language that generates client and server bindings for 10 languages. It is used in distributed computing to connect devices, mobile applications and browser clients to backend services.

Like RPC, gRPC is a form of a client–server interaction, implemented using a request–response messaging framework. This is done in the *.proto* file, a message is declared which describes the contents of the message e.g. a profile message might have a string value for name and address and an integer value for age. The request message describes the format the client wishes to parse the data.

|  |
| --- |
| message ProfileRequest {  string name = 1;  string address = 2;  int32 age = 3;  } |

Figure : gRPC message request example

The variable integer values in figure 1 on the previous page do not indicate the value of the variable, instead each variable in a gRPC message must have a unique identification number attached to it. The response message follows the same pattern.

To create a service in gRPC, both the request and response messages must be used;

|  |
| --- |
| service ProfileService {  rpc findProfile(ProfileRequest) returns (ProfileResponse);  } |

Figure : gRPC service definition example

Figure 2 above shows an example of a gRPC service being defined in a *.proto* file, in this example *findProfile* is the function, which is passed the *ProfileRequest* message, and then the gRPC service returns the response. The *.proto* file is then compiled into client and server stubs. The *.proto* file is similar to JSON and XML, but is more efficient due to its binary encoding, which improves the speed greatly.

# Code

In this section the code that will be implemented to call a remote gRPC function from the C# calculator will be discussed, the remote function will be implemented via a squared button that will take and input and multiply it by itself and return the result to the calculator.

## *.Proto* File

Firstly, the service is defined in the *.proto* file *square.proto*, this implementation only needs simple RPC, as there’s only a single client request and a server response when using the calculator. Figure 3 below shows the code used to define the *SquareService* in *square.proto*.

|  |
| --- |
| syntax = "proto3";  option java\_multiple\_files = true;  option java\_package = "io.grpc.examples.square";  option java\_outer\_classname = "SquareProto";  option objc\_class\_prefix = "SQP";  package com.example.grpc;  message SquareRequest {  double requestValue = 1;  }  message SquareResponse {  double responseValue = 1;  }  service SquareService {  rpc getSquare(SquareRequest) returns (SquareResponse);  } |

Figure : square.proto file code

In order to generate code from the *square.proto* file definitions, the Grpc.Tools NuGet package that contains Protocol Buffers compiler (protoc) and the gRPC protoc plugin are added. The *generate\_protos.bat* batch file is then run from the command line, the output is shown below in figure 4.

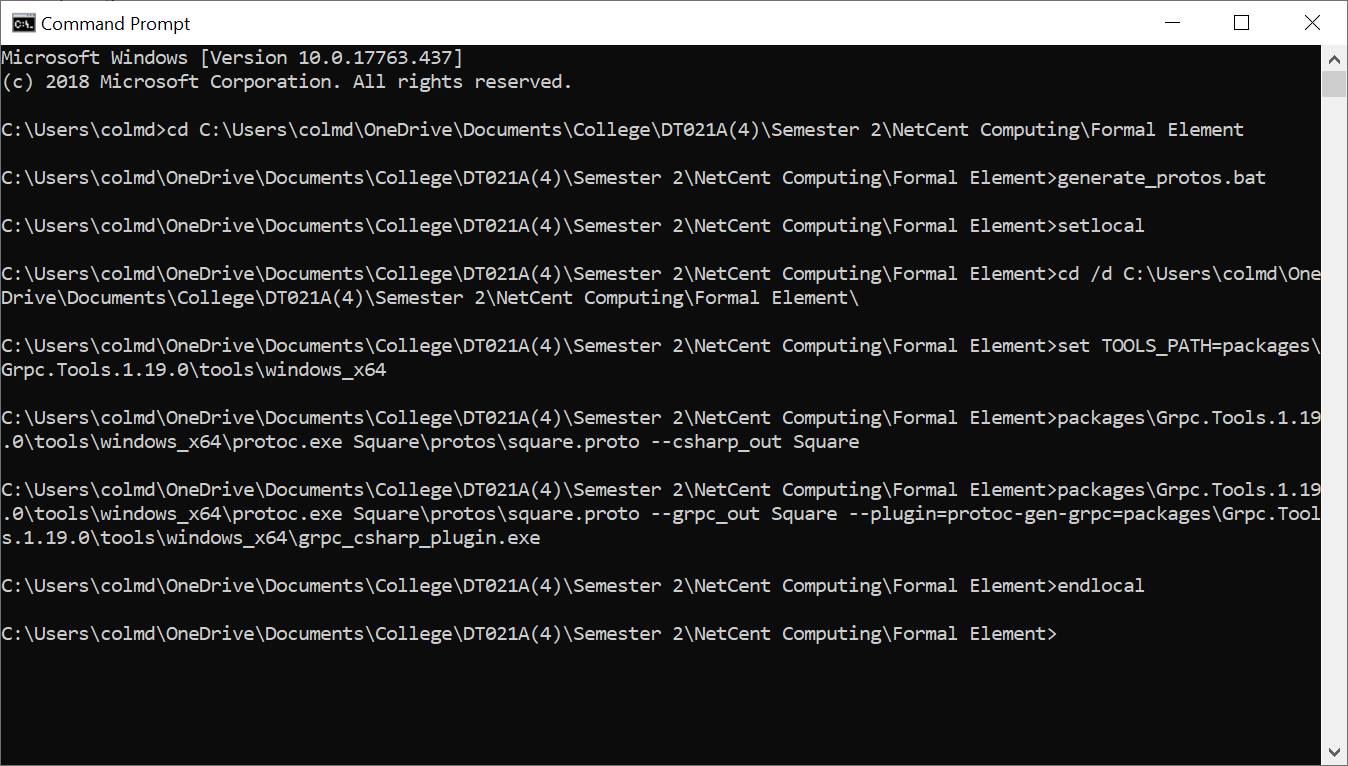


Figure : generate\_protos.bat output

Two files are generated when the batch file is run and found in the *Square/obj/Debug* directory, the first is *Square.cs* which has the necessary protocol buffer code to populate, serialize, and retrieve our request and response message types. The second file is the *SquareGrpc.cs*, this file provides generated client and server classes, including, an abstract class *SquareService.SquareServiceBase* to inherit from when defining *SquareService* service implementations, and a class *SquareService.SquareServiceClient* that can be used to access remote *SquareService* instances. The solution is built, and its output is in the form if a DLL, *Square,dll* which is used by both the client and server.

## Server

A project is added to the *GrpcCalculator* solution using .NET 4.5 framework, all the previous NuGet packages are added as well, along with a reference to the *Square.dll*. There are two parts to making our *SqaureService* service do its job, firstly implementing the service functionality by inheriting from the base class generated from our service definition, and secondly running a gRPC server to listen for requests from clients and return the service responses. Figure 5 shown below shows the code used to implement the *SquareService* functionality in the *SquareServiceImpl.cs* file.

|  |
| --- |
| using Com.Example.Grpc;  using Grpc.Core;  using System.Threading.Tasks;  using System;  namespace SquareServer  {  // SquareServiceImpl provides an implementation of the SquareService service  public class SquareServiceImpl : SquareService.SquareServiceBase  {  public override Task<SquareResponse> getSquare(SquareRequest request, ServerCallContext context)  {  var opResult = request.RequestValue \* request.RequestValue;  Console.WriteLine("|Client: " + context.Peer + " |Value: " + request.RequestValue + " |Result: " + opResult);  return Task.FromResult(new SquareResponse { ResponseValue = opResult });  }  }  } |

Figure 5: SquareServiceImpl.cs code

As you can see, our server *SquareServer* has a *SquareServiceImpl* class that inherits from the generated *SquareService.SquareServiceBase*. *SquareServiceImpl* implements all our service method, *getSquare*, which gets a *SquareRequest* from the client and returns the corresponding squared value in a *SquareResponse*.

The next part is the creation of the gRPC Server that will listen for client requests. Firstly, the port and IP are specified by creating constant variables for them and then they are used for the server configuration. The *BindService* method is used to link the server to the implementation file *SquaredServiceImpl*. The new *ServerPort* object takes the previously declared host and port numbers and adds security, in this instance none is added. The Server is then started listening on the specified port listening for client requests, until it is stopped by pressing any key.

Figure 6 on the next page shows the full code used to create the gRPC server *SquareServer*, in the *Program.cs* file.

|  |
| --- |
| using System;  using Grpc.Core;  using Com.Example.Grpc;  namespace SquareServer  {  class Program  {  const string Host = "localhost";  const int Port = 50051;  public static void Main(string[] args)  {  // Build a server  var server = new Server  {  Services = { SquareService.BindService(new SquareServiceImpl()) },  Ports = { new ServerPort(Host, Port, ServerCredentials.Insecure) }  };  // Start server  server.Start();  Console.WriteLine("SquareServer listening on port " + Port);  Console.WriteLine("Press any key to stop the server...");  Console.ReadKey();  server.ShutdownAsync().Wait();  }  }  } |

Figure 6: Program.cs server creation

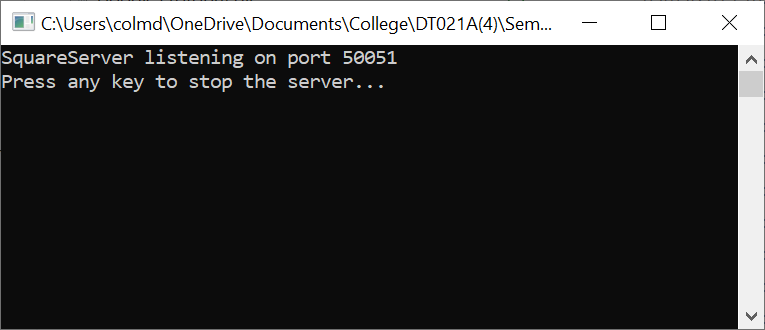
The sever is then compiled and built. To run the server the *SquareServer.exe* application file that can be found in the *SquareServer\bin\Debug* directory is run, figure 7 to the left shows the output when the server is run, it is listening for client requests on port 50051.

Figure : gRPC server running

## Client

To call service methods, we first need to create a client object (also referred to as stub for other gRPC languages). The client makes requests to the server, by passing the required parameters and then receiving the result. The client is developed similarly to the server by referencing the *Square.dll*, and all the relevant gRPC assemblies.

Firstly, a gRPC client channel must be created that will connect to gRPC server, the channel credentials must match the servers. Then, an instance of *SquareServer.SquareServerClient* class is created by passing the channel as an argument. Figure 8 below shows the full client code, from the *ColmSquareClient.cs* file. The channel is created inside the *getSquare* function with the same IP, port and security as the server.

|  |
| --- |
| using System;  using Grpc.Core;  using Com.Example.Grpc;  namespace SquareClient  {  public class ColmSquareClient  {  const string Host = "localhost";  const int Port = 50051;  public static void Main(string[] args)  {  //Do Nothing  }  public double getSquare(double value)  {  // Create a channel  var channel = new Channel(Host + ":" + Port, ChannelCredentials.Insecure);  // Create a client with the channel  var client = new SquareService.SquareServiceClient(channel);  // Create a request  var request = new SquareRequest{  RequestValue = value  };  // Send the request  var response = client.getSquare(request);  return response.ResponseValue;  }  }  } |

Figure 8: ColmSquareClient.cs client creation

Accessing the *SquareServiceClient* method must be accessed by passing the *channel* variable as a parameter. The input double *RequestValue* that was created in the *square.proto* file is made equal to the function double variable *value* and is then passed to a *SquareRequest* class.

The request is now defined and can be used by the *getSquare* method in the *SquareServiceClient*, this method returns the call to the server implementation script which will return the squared value. When the client code is compiled and built it is output as a DLL, *SquareClient.dll*.

## Calculator

The next part of the code is to implement the gRPC remote function in the C# calculator.

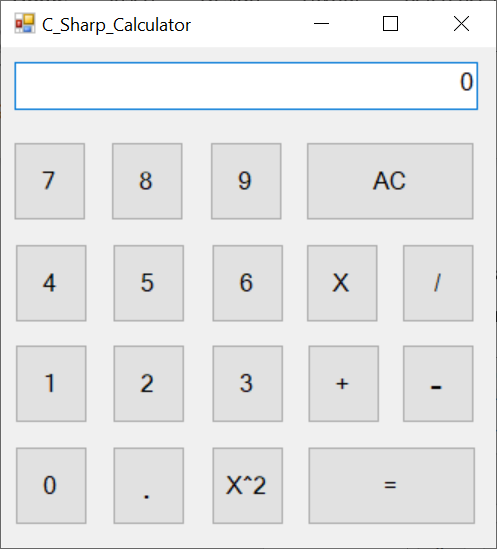
The C# calculator is created using a Visual Studio 2019 C# Windows Form application, the calculator was designed as shown to the left in figure 9. The calculator functionality is based off the apple iPhone calculator, the X2 button is added to the calculator that will call the gRPC remote function.

Figure : C# Calculator Design

The full code for the calculator is available in the appendix.

In order to call the gRPC remote function the *SquareClient.dll* must first be referenced as shown below in figure 10.

|  |
| --- |
| using SquareClient; |

Figure 10: C# sharp calculator referencing the SquareClient.dll

In order to call the remote function when the X2 button is clicked, the code must be added to the *Square\_Click* function, an object called *gRPC* of type *ColmSquareClient* is created as shown below in figure 11.

|  |
| --- |
| private void Square\_Click(object sender, EventArgs e)  {  ColmSquareClient gRPC = new ColmSquareClient();  newVal = Convert.ToDouble(T.Text);  runningTotal = gRPC.getSquare(newVal);  newVal = 0;  T.Text = Convert.ToString(runningTotal);  firstFlag = true;  firstDec = true;  add = false;  sub = false;  multiply = false;  div = false;  } |

Figure 11: C# Calculator code for square button

When the X2 button is clicked, the value that is displayed on the calculator is converted to a double and stored in the *newVal* variable. The *runningTotal* is the set equal to the result of the *getSquare* gRPC remote function when *newVal* is passed to it. *runningTotal* is then converted to a string and displayed to the calculator screen.

The function calls *SquareClient* by passing the double value to it, the *SquareClient* then sends a request to the *SquareServer* that is running, the *SquareServiceImpl* then takes in the passed value and multiplies it by itself and returns the value as a response, the response is then sent back to the calculator from *SquareClient*.

A video of the calculator’s operation is available at:

<https://youtu.be/yrhdo55p4PU>

The full code and solution is available on my GitHub at:

# Conclusion