

**gRPC Formal Report**

by

Sajjad Ullah, C17344483

This Report is submitted in partial fulfilment of the requirements of the Honours Degree in Computer & Communications Engineering (DT021A) of TU Dublin

Module: NETWORK CENTRIC COMPUTING, COMP 4600

Lecturer: Lynch, Raymond

May 9th, 2022

Contents

[Introduction background discussion 3](#_Toc102767325)

[Comparing network protocol stack of RPC and gRPC. 5](#_Toc102767326)

[Task 5](#_Toc102767327)

[Creating a gRPC client/server application 6](#_Toc102767328)

[Server 6](#_Toc102767329)

[Client 8](#_Toc102767330)

[Implementing gRPC button on Calculator 10](#_Toc102767331)

[Project flowchart 12](#_Toc102767332)

[Conclusion 13](#_Toc102767333)

[References 13](#_Toc102767334)

[Appendix A: GRPC server code 14](#_Toc102767335)

[Sqr.proto file 14](#_Toc102767336)

[squareService.cs file 14](#_Toc102767337)

[Appendix B: GRPC client code 15](#_Toc102767338)

[Sqr.proto file 15](#_Toc102767339)

[Program.cs file 15](#_Toc102767340)

[Appendix C: C# calculator implementing gRPC 16](#_Toc102767341)

[Form1.cs file 16](#_Toc102767342)

[Figure 1 NuGet Packages for server 6](#_Toc102767311)

[Figure 2 Properties tab for sqr.proto file [server side] 7](#_Toc102767312)

[Figure 3 Client project template 8](#_Toc102767313)

[Figure 4 NuGet Packages for client 9](#_Toc102767314)

[Figure 5 Properties tab for sqr.proto file [client side] 9](#_Toc102767315)

[Figure 6 server listening to port URL address 10](#_Toc102767316)

[Figure 7 gRPC output type for DLL file 11](#_Toc102767317)

[Figure 8 gRPC button for x^2 operation 11](#_Toc102767318)

[Figure 9 DLL file added 11](#_Toc102767319)

[Figure 11 gRPC testing button x^2 with input value 5 12](#_Toc102767320)

# Introduction background discussion

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 HTTP/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 variants, 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 variant, 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]

# Comparing network protocol stack of RPC and gRPC.

**Speed**: Traditional RPC runs on top of the transport layer, either TCP or UDP. From the beginning RPC was designed to be efficient. With gRPC inheriting many of the functionality of traditional RPC, it also uses the HTTP/2 and TCP layer. The structure of the protocol itself is lean, with the processing occurring at the marshaling and unmarshaling stage. This would mean that it does not require extra processing power[5]. This makes gRPC very efficient by taking advantage of the newer HTTP/2 standard.

**Data efficiency**: HTTP/2 enables highly effective, efficient uses of network resources utilizing methodologies as defined under RFC 7540. The new framing in HTTP/2 allows for decreased latency on the wire, higher data compression, and effective minifying by reducing the total amount of code without reducing greater functionality[6].

A link to the RFC 7540, Hypertext Transfer Protocol Version 2 (HTTP/2) specification can be found:

<https://datatracker.ietf.org/doc/html/rfc7540>

**Performance**: By building gRPC within the framing of HTTP/2, it gets the benefit of RPC magnified by the gains in HTTP/2, meaning smaller data with equal functionality. One important feature of HTTP/2 its bi-directional streaming capabilities in the transport specification, something that gRPC takes advantage of to minimize waste data and decrease overall latency[7]. What you end up with is a lean platform using a lean transport system to deliver lean bits of code — an overall decrease in latency, size, and demand that is noticeable and enables smaller, less adept hardware the same processing power of larger, more powerful contemporaries[7][8].

# Task

The objective of this assignment is to implement a gRPC remote function on a C# calculator application which will return a value to be displayed on the calculator screen.

The application is based upon a Windows Forms calculator taking user input via GUI and passing it to the client.

The gRPC client code will be converted into a DLL that can be accessed by the calculator application.

We will be building a x^2 button.

# Creating a gRPC client/server application

To start, create a new project based on the template for creating a “gRPC ASP.NET Core Service using .NET Core”

## Server

Labelling the project name as “GrpcServer” here we will develop the server side.

First install the NuGet Packages shown in figure 1 for the server.

Graphical user interface, application

Description automatically generated

Figure 1 NuGet Packages for server

Under the folder *Protos*, the *.proto* file is used to define the contract: i.e., which methods does the service provide, and what are the request/response message exchanged by client and server. We created a new .proto file called “sqr.proto”. here we define the service “RetSquare” (shorthand for return squared) and message.

option csharp\_namespace = "GrpcServer";

service Square {

// creating a service for the x^2 function

rpc RetSquare(SquareRequest) returns (SquareReply){};

}

message SquareRequest { // request message passed to square function

float inval = 1; // the values assigned here represents the order in which the values come through

}

message SquareReply { // this is returned when square function is called

float outval = 1;

}

Then right click on this protos file and open the properties to ensure it builds as a “Protobuf compiler” under *Build Action*. Also, that *gRPC Stub Classes* is set to “Server only”.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 2 Properties tab for sqr.proto file [server side]

Next, we will add a new file to the Services folder. Right Click on the folder -> Add-> New Item and select the “Class” option. We label this file “squareService.cs”. These classes are generated by the gRPC tooling, based on the .proto file, providing all the functionality needed to send/receive messages so we only have to consider the method implementation. This will contain all the necessary protocol buffer code to satisfy the reply and request types.

Once opened, ensure the class inherits from the protos file. To do this we inherit from the name of the service in the protos file which was *Square*.

Use the syntax:<serviceName>.<serviceName>Base

namespace GrpcServer.Services

{

public class squareService : Square.SquareBase

{

private readonly ILogger<squareService> \_logger;

public squareService(ILogger<squareService> logger)

{

\_logger = logger;

}

**Implementing the RetSquare function:**

public override Task<SquareReply> RetSquare(SquareRequest request, ServerCallContext context)

{

return Task.FromResult(new SquareReply { Outval = request.Inval \* request.Inval }); // square function is implemented

}

}

}

Once complete build the server and now move onto the client.

## Client

The client makes requests to the server, passing the required parameters and receiving the result. The client is developed similarly to the server.

Right Click on the project-> Add new project and create using “Console App (.NET Core) “ Note: ensure its C# compatible as there may be multiple options as shown in figure 3.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 3 Client project template

Label this project “GrcpClient”.

First install the NuGet Packages shown in figure 4 for the client.

Graphical user interface, text

Description automatically generated

Figure 4 NuGet Packages for client

Under the Protos folder copy the “sqr.proto” file over to this project. Then open the copied file and change the namespace to that matching the name of the project.

option csharp\_namespace = "GrpcClient";

Then right click on this protos file and open the properties to ensure it builds as a “Protobuf compiler” under *Build Action*. Also, that *gRPC Stub Classes* is set to “Client only”.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5 Properties tab for sqr.proto file [client side]

Next open the file “Program.cs”, inside the main a new channel is created. The channel in gRPC is made to define a connection to a remote server.

During first test, the client is built with output for console to test both client and server programs by passing in a number at Inval e.g. “Inval = 5“.

Once tested, it can be turned into a function called “RetSquare” with a input type float and return value type float also.

namespace GrpcClient

{

public class ClientProgram

{

public float RetSquare(float x)

/\*public static void Main(string[] args) for console\*/

{

//creating a GrpcChannel using the URL of our service

var channel = GrpcChannel.ForAddress("https://localhost:5001");

var client = new Square.SquareClient(channel);

//now we can use the client instance to call the RetSquare method

// passes input value of function to create a new request

var request = new SquareRequest { Inval = x };

// use request to return reply from server

var response = client.RetSquare(request);

// Console.WriteLine("Answer:" + response.Outval);

// Console.ReadLine();

return response.Outval; // give calculator app the x^2 answer

}

}

}

The channel is created using the URL address given by the server that is listening on http://localhost:5001 as shown in figure 6.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 6 server listening to port URL address

It will be easier to convert gRPC client code into a DLL that can be accessible from the calculator. To convert the client code to a DLL we rebuild it as a class library inside of which is the class definition. Open the properties tab and se as shown below:

Graphical user interface, application

Description automatically generated

Figure 7 gRPC output type for DLL file

# Implementing gRPC button on Calculator

Using a C# calculator application created during the labs we will add a button for a x^2 remote operation.

A picture containing furniture, indoor, file, screenshot

Description automatically generated

Figure 8 gRPC button for x^2 operation

Inside the GrpcClient project navigate to the bin->debug> directory and copy the “GrpcClient.dll” into the C# calculator project bin->debug-> directory.

Then, open the project and add the .dll file into the dependencies folder.

Text

Description automatically generated

Figure 9 DLL file added

Once added open “Form1.cs” and add the name of the dll file to the top like so:

using GrpcClient;

Then navigate down to the button event handler and create a class object to call the function RetSquare().

private void button17\_Click(object sender, EventArgs e)// x^2 button

{

ClientProgram obj = new ClientProgram();//creating a class object

float num;

num = Convert.ToSingle(textBox1.Text);

//call the function as you would to a class function

float sqrVal = obj.RetSquare(num);

textBox1.Text = Convert.ToString(sqrVal);

result = sqrVal;//running total

}

When building for the first time there were issues encountered during build time, these were fixed by adding the Google.Protobuf, Grpc.Net.Client & Grpc.Tools packages to the project.

The value 5 was entered into the calculator and the output result is not only correct but also tells us the gRPC button is working as intended.

Graphical user interface

Description automatically generated with medium confidence

Figure 11 gRPC testing button x^2 with input value 5

# Project flowchart

Diagram

Description automatically generated

**gRPC connection Request/Response:**

Diagram

Description automatically generated

# Conclusion

In conclusion we have successfully implemented a gRPC remote button function in a C# calculator application. We learned that gRPC has many of the same functionality of traditional RPC such as operating on TCP layer but was designed to take advantage of newer standards such as HTTP/2 and made to be leaner, for example by having the processing occur in the marshaling and unmarshaling stage. What we end up with is a platform using a lean transport system with an overall decrease in latency, size, and demand that is noticeable and enables smaller, less adept hardware the same processing power of larger, more powerful contemporaries.

# 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].

[5]"gRPC - A Modern Framework for Microservices Communication | Capital One", *Capital One*, 2020. [Online]. Available: https://www.capitalone.com/tech/software-engineering/grpc-framework-for-microservices-communication/. [Accessed: 06- May- 2022].

[6]"Routing HTTP/2 and gRPC Traffic to Apps | Cloud Foundry Docs", *Docs.cloudfoundry.org*. [Online]. Available: https://docs.cloudfoundry.org/devguide/http2-protocol.html. [Accessed: 06- May- 2022].

[7]P. Cruickshank, "The Mysterious Gotcha of gRPC Stream Performance | Ably Blog: Data in Motion", *Ably Blog: Data in Motion*. [Online]. Available: https://ably.com/blog/grpc-stream-performance. [Accessed: 06- May- 2022].

[8]"Core concepts, architecture and lifecycle", *gRPC*, 2022. [Online]. Available: https://grpc.io/docs/what-is-grpc/core-concepts/. [Accessed: 06- May- 2022].

# Appendix A: GRPC server code

## Sqr.proto file

syntax = "proto3";

option csharp\_namespace = "GrpcServer";

service Square {

// creating a service for the x^2 function

rpc RetSquare(SquareRequest) returns (SquareReply){};

}

message SquareRequest { // request message passed to square function

float inval = 1; // the values assigned here represents the order in which the values come through

}

message SquareReply { // this is returned when square function is called

float outval = 1;

}

## squareService.cs file

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.Extensions.Logging;

using Grpc.Core;

namespace GrpcServer.Services

{

//The generated squareService class contains the implementation of the service

public class squareService : Square.SquareBase // inherits abstract base class

{

private readonly ILogger<squareService> \_logger;

public squareService(ILogger<squareService> logger)

{

\_logger = logger;

}

public override Task<SquareReply> RetSquare(SquareRequest request, ServerCallContext context)

{

return Task.FromResult(new SquareReply { Outval = request.Inval \* request.Inval }); // square function is implemented

}

}

}

# Appendix B: GRPC client code

## Sqr.proto file

Only difference is the namespace for the client project.

option csharp\_namespace = "GrpcClient";

## Program.cs file

using System;

using System.Threading.Tasks;

using Grpc.Net.Client;

namespace GrpcClient

{

public class ClientProgram

{

public float RetSquare(float x)

/\*public static void Main(string[] args) for console\*/

{

//creating a GrpcChannel using the URL of our service

var channel = GrpcChannel.ForAddress("https://localhost:5001");

var client = new Square.SquareClient(channel);

//now we can use the client instance to call the RetSquare method

var request = new SquareRequest { Inval = x }; // passes input value of function to create a new request

var response = client.RetSquare(request); // use request to return reply

// \*\*\*\*\* needed for console app

// Console.WriteLine("Answer:" + response.Outval);

// Console.ReadLine();

return response.Outval; // give calculator app the answer

}

}

}

# Appendix C: C# calculator implementing gRPC

## Form1.cs file

At the top:

using GrpcClient;

Inside the x^2 button event handler:

private void button17\_Click(object sender, EventArgs e)// x^2

{

//create an object to be able to use remote function

ClientProgram obj = new ClientProgram();

float num;

num = Convert.ToSingle(textBox1.Text);

float sqrVal = obj.RetSquare(num);//calling remote gRPC function

textBox1.Text = Convert.ToString(sqrVal);

result = sqrVal;//running total

}