B 

**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

22 April 2019

Lecturer: Dr Ray Lynch

**Contents**

[Introduction 2](#_Toc6583777)

[Objective 3](#_Toc6583778)

[Code 4](#_Toc6583779)

[Server 5](#_Toc6583780)

[Client 6](#_Toc6583781)

[Figure 1 Server Running 6](file:///C:\Users\Keyser%20Soze\Net_Comp\gRPC\Jack%20Harding%20gRPC%20Report.docx#_Toc6583793)

# Introduction

1. A background discussion on gRPC. Include a comparison with traditional RPC.

Google developed it for object serialisation and is a mechanism in gRPC for generating language specific code. Similar to RPC, the objects are defined in a definition file where the input and output parameters are defined, interfaces are replaced by messages in gRPC and are used in .proto files. Inside an RPC interface is a function which specifies input, output, and the parameters each take; the same is done with a proto file, a message is declared which describes the contents of the message e.g. a car message might have a string value for the model and a integer value for the price. The request message 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 NewRequest {

int32 price = 1;

string model = 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.

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

service NewService {

rpc MyFunc(NewRequest) returns (NewResponse);

rpc AnotherFunc(AnotherRequest) returns (AnotherResponse);

}

MyFunc is the function which is passed NewRequest this gRPC service returns the response. The code above is written in the .proto file which is then compiled into client and server stubs.

Similar to JSON and XML but is more efficient due to its binary 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.

# Objective

# 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 satisfy the reply and request types, second is the *SquareGrpc.cs*, this file acts like a stub for both client and server and has the necessary classes. When this project is built its output is a DLL which is used by both client and server.

## 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*, 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 and adds security (none in this case).

public class SquaredServiceImpl : SquareService.SquareServiceBase

{

public override Task<SquareReply> RetSquare(

SquareRequest request,

ServerCallContext context

) {

return Task.FromResult(

new SquareReply {

Outval = request.Inval \* request.Inval

});

}

}

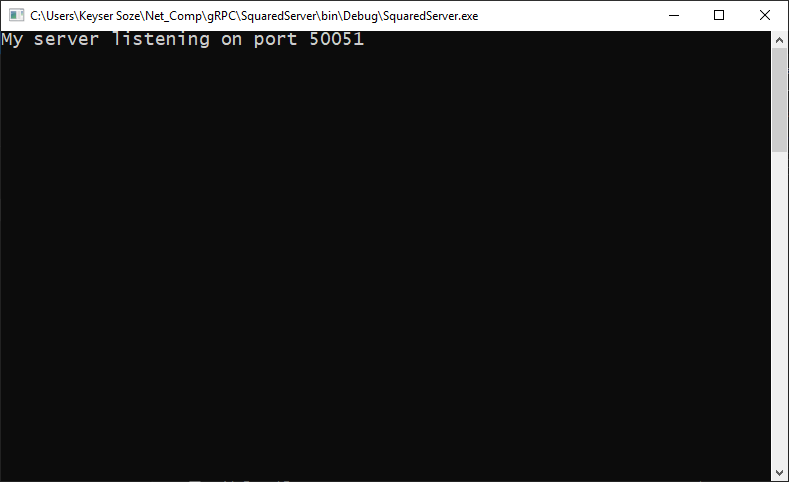


Figure 1 Server Running

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 jijdajavnj, 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 assemblies. To begin, the client is built as an executable to test both client and server programs. The host and port numbers must be identical to 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 (none in this case).

var client = new SquareService.SquareServiceClient(channel);

Accessing the SquareServiceClient method must be accessed by passing a parameter as it throws an exception for 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* classwhich 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 bhefbhbh

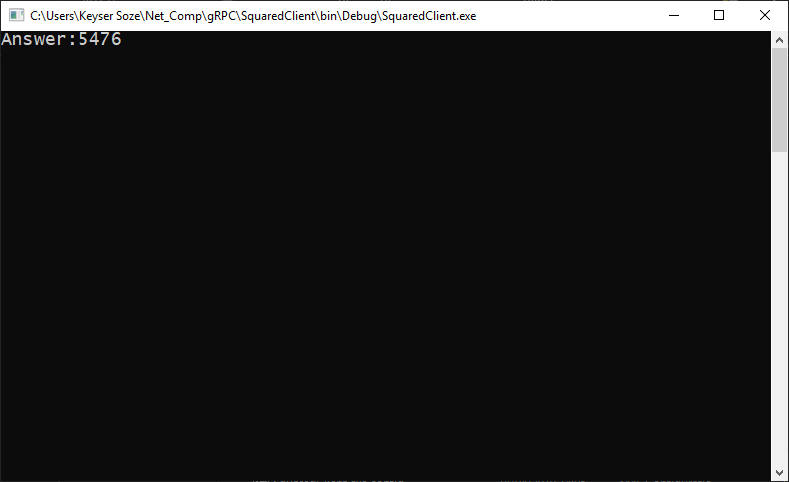


Figure 2 Reply from Server

## Calculator

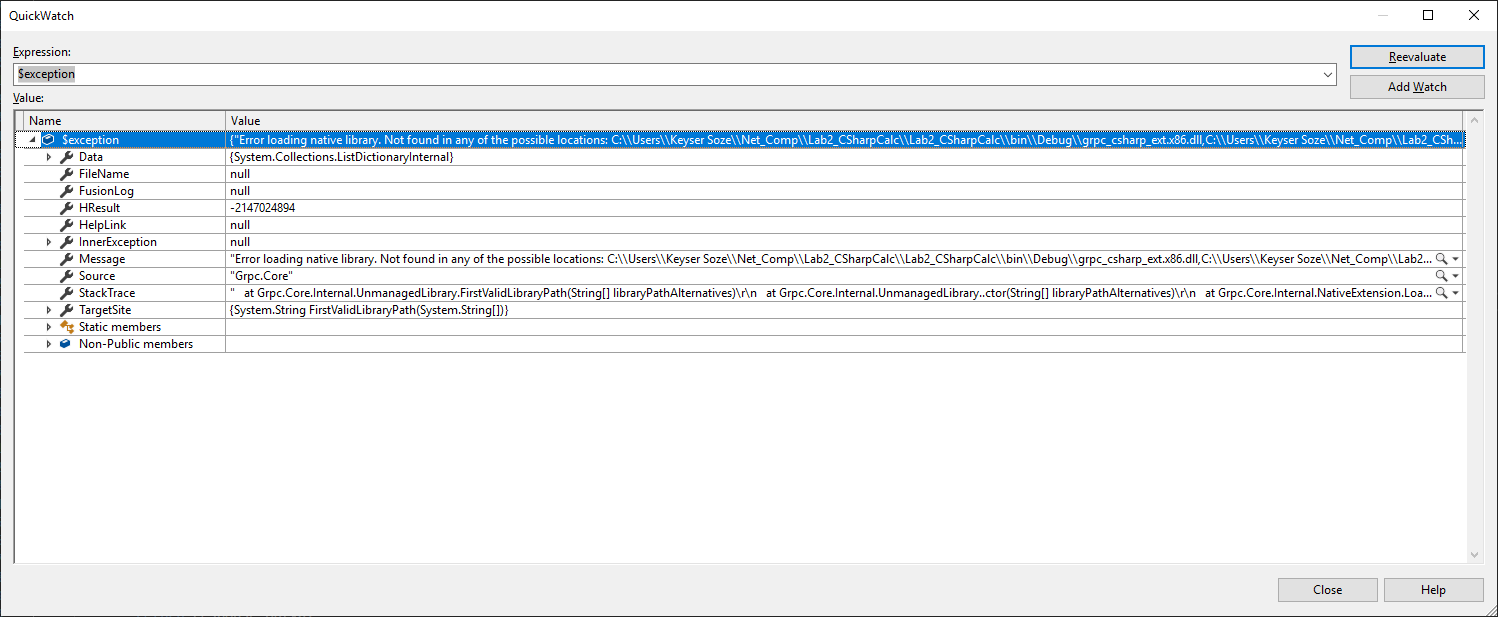
The calculator

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Code | Description | Project | File | Line |
| CS0012 | The type 'Nullable<>' is defined in an assembly that is not referenced. You must add a reference to assembly 'netstandard, Version=2.0.0.0, Culture=neutral, PublicKeyToken=cc7b13ffcd2ddd51'. | SquaredClient | gRPC\SquaredClient\ClientProgram.cs | 30 |

System.IO.FileNotFoundException

HResult=0x80070002

Message=Error loading native library. Not found in any of the possible locations: C:\Users\Keyser



1. Add a button to your C# calculator
2. Calls a gRPC remote function and displays the return value on the calculator screen.
3. It is up to you to decide what the remote function will do.

* A detailed description of your particular implementation of the code above.
* You should include an explanation of the particular code snippets related to gRPC.
* Also include flowcharts etc. that will demonstrate your understanding of the technology.

# Conclusion