Network Centric and Distributed Computing Formal Assignment: gRPC with C# and .NET Framework

Pierce Lowe, C18319993

07/05/2022

DT021A/4 | COMP4600

Table of Contents

Introduction

The goal of this project is to utilise Google's remote procedure call (RPC) framework, gRPC, to make remote function calls in a simple GUI application written using C# and .NET Framework. This report will examine what RPC is followed by taking a look at what gRPC adds on top of the basic RPC mechanism and finally reviewing the implementation in code and how it was added to the GUI application, in this case a basic calculator built using Windows Forms.

The project source code can be found here: https://github.com/Pieloaf/COMP4600_Formal_gRPC

Background

RPC

RPC is a mechanism used to allow clients to call functions on a remote server without the need for the developer to explicitly define details about the network communication for each call, such as the servers address, the listening port, etc. In code, RPC makes remote function calls look the same as any local function call, however a lot more is happening behind the scenes.

Calling a function on a remote machine faces a number of difficulties such as identifying the process to be executed on the host, varying architectures between client and host, loss or reordering of messages, among others [1]. RPC libraries attempt to abstract these complications away from the developer. When a remote function call is made, the function parameters are sent to a client stub which has a template of the remote function. This client stub then marshals (translates) the function parameters so that they are understandable by the server stub. The client stub then makes a request to the remote server, once the client stub receives a response from the server it unpacks the data from the server back to a data format understandable by the client and returns the result to the client program. This client side function call may be made asynchronous to prevent the client program from blocking while it waits for a response from the server.

On the server side, once the client stub makes the request to the server and the data is received by a sever stub which unpacks it into a format that the server side program can use, the function is then executed by the remote server and the response is returned to the client by passing it through the server stub once again to marshal the data in preparation for the client [1].

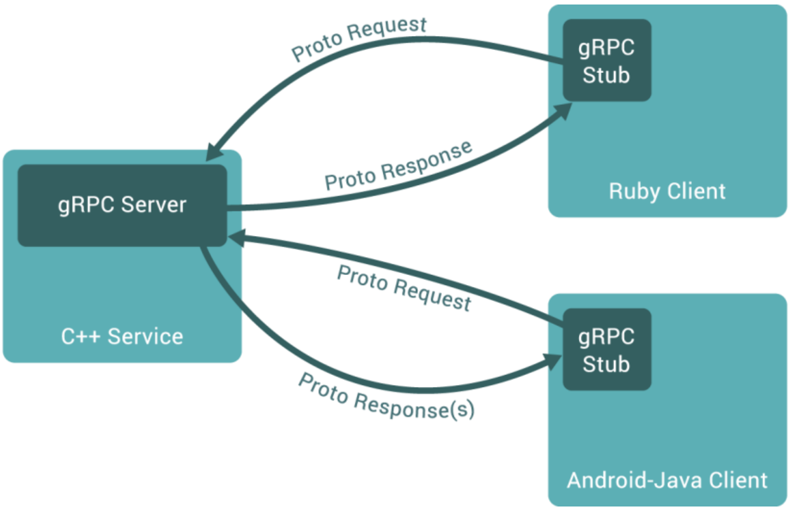

Figure 1: Flowchart illustrating that the client/server communication is handled by the stubs rather than the program itself, allowing for language agnosticism. Flowchart is for gRPC but same basic principal applies to all RPC implementations [2]

Stubs and IDLs

Typically the aforementioned stub files for the client and server are not written by hand by the developer. They are constructed automatically using an interface definition language (IDL). IDL is a generic term for a language that allows programs of different languages or systems with differing architectures to communicate. The format can vary between RPC library implementations however the idea remains the same. IDL files contain a number of different attributes. They typically begin with some metadata such as a UUID to identify the interface as well as an interface version, they may also contain other information such as binding handles or endpoints. After that the services are defined with function prototypes containing the function name, input parameters and return types [3].

The IDL file is then passed through an IDL compiler which then generates the stub files in the desired language for both server and client [3] [4]. These stub files are then referenced by the client and server process to handle the RPC requests.

gRPC

As mentioned previously, RPC is simply a mechanism. This means it is not a set communication standard or framework as such, this allows for anyone to create their own RPC framework or library. Previously Sun created Sun RPC and the Open Software Foundation commissioned DCE RPC. Both of which run directly on top of the UDP and TCP transport protocols [1]. In 2015, Google released their own open source RPC framework known as gRPC. gRPC is another implementation of the RPC mechanism, however where it differs from other implementations is that gRPC is designed with cloud infrastructure, microservice architecture and scalability in mind, meaning that rather than a client calling a remote procedure directly on a remote server, the request is made on a remote service. The request is received first by a load balancer which then determines where to handle the request [1].

Following cloud services design, the requests are not handled by a single server but by a number of servers which can be dynamically created and destroyed to meet demand. These servers are created to run the process and subsequently destroy it once the process is run. These server are not physical servers but highly optimised isolated virtual environments known as containers, only containing the necessary system components to run the process, this allows for high speed creation and destruction of these containers. The industry standard containerisation platform is Docker with Kubernetes typically running on top of Docker as a system for deploying and managing all these containers.

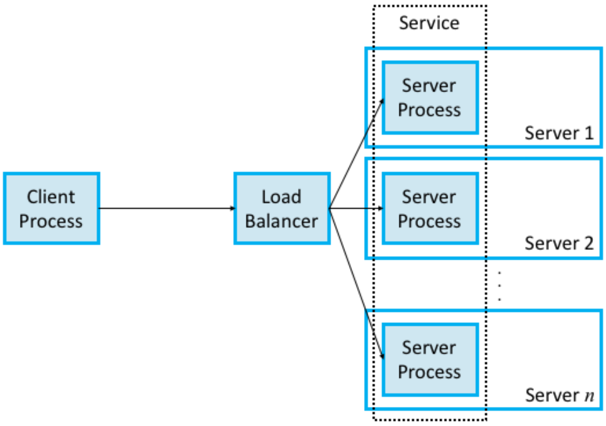

Figure 2: Using RPC to invoke a scalable cloud service [1]

gRPC utilises TCP/IP to handle connectivity and reliability as well as implementing a layer of security through the use of TLS running on top of TCP. Although other RPC mechanisms typically run directly on top of the transport layer, gRPC runs on top of HTTP/2 protocol, this is quite a big difference from other implementations as it adds a whole other layer of abstraction. Running on top of HTTP/2 allows for more efficient data encoding and for multiplexing multiple function calls on a single TCP connection [1], as well as near universal support for a lot of third party-tools for load balancing, encryption, authentication, etc. since HTTP is the application layer protocol of the web.

On the client side, gRPC connections are encapsulated inside channels and abstracted away from the developer. These channels can contain a number of connections to a server at a specified host and port [5] and can be used to configure settings for the underlying connections such as enabling or disabling data compression [5]. It is also recommended to reuse channels when possible rather than creating a new channel for each connection [6].

As well as the typical unary RPC communication system, where the client and host will only send one message at a time, gRPC provides three variations of streaming mechanisms. The first is server streaming in which the server responds to a message with a stream of messages. The second is client streaming where the client sends a stream of messages to the server before it awaits a response and finally bidirectional streaming where the client streams messages to the server and receives a stream of messages from the server in response, however bidirectional streams are independent of each other and so the server does not need to wait for the end of the message stream before replying. gRPC also ensures correct message ordering for each of these stream types [5].

Methodology

The following section will look at how gRPC was integrated into a simple calculator application. Most of the calculator functions are run locally however the three basic trigonometric functions are executed on the gRPC server.

Protocol Buffers (protobuf)

As mentioned previously IDLs are used to define the server and client stub files to be generated which handle the marshalling of data on both the client and server so the data is understandable at each end. gRPC uses Googles Protocol Buffer IDL [3], known as protobufs, to generate the stub files. In the protobuf file the language namespaces are defined along with the message structures for requests and responses as well as the services and its functions (See Appendix A). From this file the server and client stub files are created across two files.

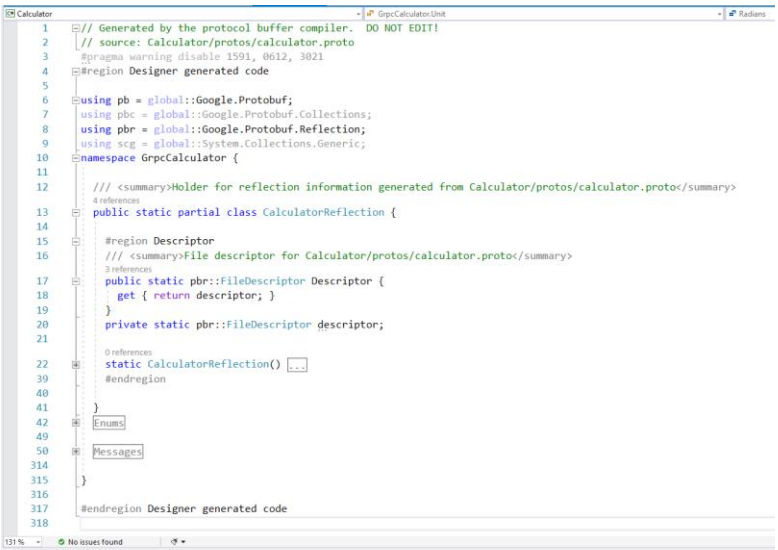

Figure 3: One of the two generated stub files from the protobuf file for C#

Server implementation

The gRPC server consists of two files, one which contains the main program that creates the listening server while the other implements the calculator service functions. Inside the main function for the server, the server object is created defining the services, in this case a single calculator service, and the hostname and the port to listen on along with any security credentials if applicable. The server is then started and run until it receives user input (See Appendix B).

The calculator service implementation defines a class which inherits from the CalculatorServiceBase in the stub file generated from the protobuf. Within the service class the three basic trigonometric functions are implemented, each of which returns a “Task” object. Tasks are asynchronous functions i.e. non-blocking functions and so allows the server to handle other requests and operations while the requested function is being carried out. Once the asynchronous task is complete it returns the result which is passed to the stub file to respond to the client with. Each of the three functions checks the unit of the input i.e. degrees or radians, and adjusts the input accordingly. On the asynchronous task is completed, a MathResponse object, as defined in the protobuf, is returned with the sine, cosine or tangent of the value as requested by the client (See Appendix C).

Client Implementation

The client program begins by defining a Response struct, this is implemented to handle errors communicating with the gRPC server. It contains a valid bool to indicate to the program utilising the client DLL if the response received is a valid response or an error message. It also contains an “ans” string containing the response to the function, or the error message in case of an error (See Appendix D).

Following that, a TrigCalculator class is defined, this initialises variables to store the host and port of the server to be communicated with, a client object of type CalculatorServiceClient from the client stub file and a channel object of type Channel from the gRPC C# library. As discussed earlier it is best practice to reuse gRPC channels where possible and so a single channel is defined and created once for the client class, all gRPC requests are made through this channel and the channel is then only shut down on request by the client. The default constructor for the class sets a default hostname and port for the server, however there is also an overloaded constructor for the TrigCalculator class which takes a hostname and port as arguments to allow the default host and port to be overridden within the GUI application code rather than being required to update the gRPC client DLL manually (See Appendix E).

After the constructors, there are a number of helper functions defined. Firstly, a CreateChannel function is defined which opens the gRPC channel for communications with the server which remains open until requested. An asynchronous function, CloseChannel, is then also defined which is to be called to close gRPC the channel. Finally a private createReq function is defined which is to be called on each request to construct the TrigRequest object to be sent to the server (See Appendix E).

Finally the trigonometric functions are implemented. Each follows the same basic structure, create the request by passing the value and unit to the createReq function. The client then tries to call the remote function with the request object, if the client receives a response a new Response object is created with the valid flag set to true and the answer string set to the result returned from the server. In the case of an RPC exception the Response object is created with a false valid flag and the error string is set as the answer. The response object is then returned from the function (See Appendix F)

GUI Application

The GUI application is a simple Windows Forms .NET framework application. It consists of a grid of numbers with function buttons down the right hand side with the gRPC function buttons on the left.

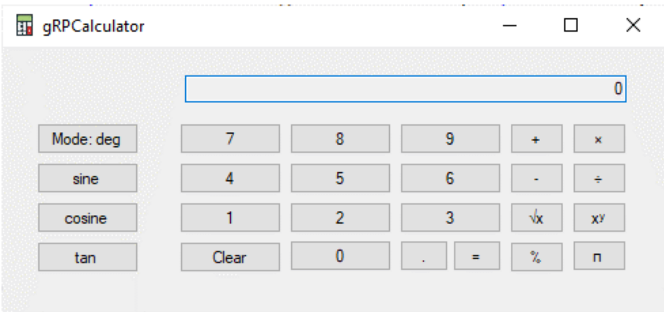

Figure 4: Calculator Application Window

Firstly a number of private variables are initialised such as setting the runningTotal to 0, the first operation to addition, initialising the gRPC client and setting a number of boolean flags. When the constructor for the application window is called, the gRPC client instance calls the previously mentioned CreateChannel function to setup the gRPC communication channel. Following that a function to be called on the window closing is setup to close the gRPC channel on the application window being closed (See Appendix H)

Next a number of helper functions are defined for handling button inputs. numberInput is a function that handles the number inputs and displays them in the text box. A functionInput function handles mathematical function execution and updating the display with the result. Finally handleTrigResponse was setup to handle the gRPC server responses by displaying the result from the server assuming it was successful or in the event of an error displaying an error message in the text box and setting an rpcError flag for the functionInput to read when handling the next input (See Appendix H).

Finally the button handler call backs were defined, the number button and function button callbacks work in the same way: the text on the button is read and passed through to numberInput and functionInput functions respectively for processing. The clear button handler resets the running total value and text box. It also sets the previous operation to addition and resets the firstFlag to its initial value of true and rpcError flag to false (See Appendix I).

The pi button calls the numberInput function again, however instead of passing the value on the button, in this case the pi symbol, it gets the value of pi from the built-in Math library which it then passes as an argument to the function. The decimal point button, first checks to make sure there is not already a decimal point entered before calling numberInput to add it to the text box (See Appendix I).

An angle unit button was added to allow the user to choose between degrees and radians for the trigonometric functions. The handler for this button checks if the current unit is degrees, if so the button text is set to “Mode: rad” and the deg flag is set to false, otherwise the button text is set to “Mode: deg” and the deg flag is set to true (See Appendix I).

Finally the trigonometric button handlers were defined, each calls their respective gRPC function and passes through the value from the text box as a double, along with the degree flag set by the angle unit button. The gRPC client DLL then handles the request as discussed previously and attempts to make a request to the server, returning the result or an error along with the validity of the response, the button handlers then pass this result to the previously discussed handleTrigResponse function to display the result and set any flags for the next input (See Appendix J).

Conclusions

The application works as expected, with the button making a function call in the gRPC calculator client DLL which then makes a remote functional call to the gRPC server. If the server is online and receives the request, it runs the process and returns the result client stub which is then handed off to the DLL once the data is unpacked. The client DLL returns the data to the GUI application in a valid response object to be output on the calculator textbox. In the event the client cannot reach the server it returns an invalid response object and displays an error message on the calculator and resets values to their starting state to avoid issues with future inputs.

RPC as a whole is an especially useful communication mechanism for remote operations that need to be carried out requiring higher performance or environments in which the server and client are tightly coupled. After working with gRPC, it’s clear to see how it offers a number of clear benefits over standard RPC mechanisms, especially the use of HTTP/2 for handling compression and managing TCP connections, as well as utilising TLS, removing the ownness from the developer to implement their own security mechanism and focus solely on the functionality of the application.

References

[1] L. Peterson and B. Davie "Remote Procedure Call" in *Computer Networks: A Systems Approach*. 6th ed. Elsevier

[2] gRPC. 2021. *Introduction to gRPC*. [online] Available at: https://grpc.io/docs/what-is-grpc/introduction/ [Accessed 7 May 2022].

[3] Docs.microsoft.com. 2022. *Interface Definition Language - gRPC for WCF Developers*. [online] Available at: https://docs.microsoft.com/en-us/dotnet/architecture/grpc-for-wcf-developers/interface-definition-language [Accessed 7 May 2022].

[4] Docs.microsoft.com. 2019. *Invoking the MIDL Compiler - Win32 apps*. [online] Available at: https://docs.microsoft.com/en-us/windows/win32/midl/invoking-the-midl-compiler [Accessed 7 May 2022].

[5] gRPC. 2021. *Core concepts, architecture and lifecycle*. [online] Available at: https://grpc.io/docs/what-is-grpc/core-concepts/ [Accessed 7 May 2022].

[6] gRPC. 2021. *Performance Best Practices*. [online] Available at: https://grpc.io/docs/guides/performance/ [Accessed 7 May 2022].

Appendices

Appendix A. gRPC Calculator Protocol Buffer

```
syntax = "proto3";

option csharp_namespace = "GrpcCalculator";

enum Unit {
    DEGREES = 0;
    RADIANS = 1;
}

message TrigRequest {
    double value = 1;
    Unit unit = 2;
}

message MathResponse {
    double answer = 1;
}

service CalculatorService {
    rpc sine(TrigRequest) returns (MathResponse);
    rpc cosine(TrigRequest) returns (MathResponse);
    rpc tan(TrigRequest) returns (MathResponse);
}
```

Appendix B. gRPC Calculator Server: Server Handler

```
using System;
using Grpc.Core;
using GrpcCalculator;

// using CalculatorServer namespace
namespace CalculatorServer
{
    // Main server program class
    0 references
    class Program
    {
        const string Host = "localhost"; // setting host variable
        const int Port = 50051;          // setting port variable

        // Main server function
        0 references
        public static void Main(string[] args)
        {
            // Create a gRPC Server object
            var server = new Server
            {
                // define servers service [server only has calculator service with the calculator service implementation]
                Services = { CalculatorService.BindService(new CalculatorServiceImpl()) },
                // setting the server hostname and listening port from the previously set variables
                Ports = { new ServerPort(Host, Port, ServerCredentials.Insecure) }
            };

            // Start server listening
            server.Start();

            // Print to console...
            Console.WriteLine("CalculatorServer listening on port " + Port);
            Console.WriteLine("Press any key to stop the server...");
            Console.ReadKey(); // wait for user input

            // shut down server on user input
            server.ShutdownAsync().Wait();
        }
    }
}
```

Appendix C. gRPC Calculator Server: Service Implementation

```
using GrpcCalculator;
using Grpc.Core;
using System.Threading.Tasks;
using System;

// using CalculatorServer namespace
namespace CalculatorServer
{
    // Calculator Service Class Implementation
    1 reference
    public class CalculatorServiceImpl : CalculatorService.CalculatorServiceBase
    {
        // async sine function that returns a Task with MathResponse
        // on function completetion the task returns the MathResponse object
        2 references
        public override Task<MathResponse> sine(TrigRequest request, ServerCallContext context)
        {
           double val = request.Value; // store request value in val

            // check if request units are degrees
            if (request.Unit == Unit.Degrees)
            {
                //if degrees, convert to radians
                val = ( request.Value * (Math.PI)) / 180;
            }
            // return Task with MathResponse object with Sin of val
            return Task.FromResult(new MathResponse { Answer = Math.Sin(val) });
        }

        // async cosine function that returns a Task with MathResponse
        // on function completetion the task returns the MathResponse object
        2 references
        public override Task<MathResponse> cosine(TrigRequest request, ServerCallContext context)
        {
            double val = request.Value; // store request value in val

            // check if request units are degrees
            if (request.Unit == Unit.Degrees)
            {
                //if degrees, convert to radians
                val = (request.Value * (Math.PI)) / 180;
            }
            // return Task with MathResponse object with Cosine of val
            return Task.FromResult(new MathResponse { Answer = Math.Cos(val) });
        }

        // async tan function that returns a Task with MathResponse
        // on function completetion the task returns the MathResponse object
        2 references
        public override Task<MathResponse> tan(TrigRequest request, ServerCallContext context)
        {
            double val = request.Value; // store request in val

            // check if request units are degrees
            if (request.Unit == Unit.Degrees)
            {
                //if degrees, convert to radians
                val = (request.Value * (Math.PI)) / 180;
            }
            // return Task with MathResponse object with Tan of val
            return Task.FromResult(new MathResponse { Answer = Math.Tan(val) });
        }
    }
}
```

Appendix D. gRPC Calculator Client: Response Struct

```
using System.Threading.Tasks;
using Grpc.Core;
using GrpcCalculator;

// using the CalculatorClient namespace
namespace CalculatorClient
{
    // Defining the Response Struct
    10 references
    public struct Response {
        // struct constructor definition
        6 references
        public Response(bool v, string a)
        {
            valid = v;  //indicates if result is valid
            ans = a;    // result
        }

        2 references
        public bool valid { get; }  // defining valid type
        2 references
        public string ans { get; }  // defining answer type

        // used in testing for easier printing of result
        0 references
        public override string ToString() => $"({ans}, {valid})";
    }
```

Appendix E. TrigCalculator Class: Constructors and Helper Functions

```
    // TrigCalculator class definition and implementation
    2 references
    public class TrigCalculator
    {
        // Attribute Initialisation
        private string Host;    // server address
        private int Port;       // server listening port
        private CalculatorService.CalculatorServiceClient client; // gRPC client object
        private Channel channel;    // gRPC channel object

        // Constructor
        0 references
        public TrigCalculator()
        {
            // Set Host and Port Variables
            Host = "localhost";
            Port = 50051;
        }

        // Overloaded Constructor
        0 references
        public TrigCalculator(string host, int port)
        {
            // Allows user to define an alternate host and port
            Host = host;
            Port = port;
        }

        // create channel function
        0 references
        public void CreateChannel()
        {
            // Create an insecure gRPC channel to a given host and port
            // these channels can have multiple underlying connections
            channel = new Channel($"{Host}:{Port}", ChannelCredentials.Insecure);

            // Create a Service Client using the previously made channel
            // Function calls are made through this client
            client = new CalculatorService.CalculatorServiceClient(channel);
        }

        // Async Close channel function
        0 references
        public async Task CloseChannel()
        {
            // When this function is called the gRPC channel will shutdown
            await channel.ShutdownAsync();
        }

        // helper function to create a request object
        3 references
        private TrigRequest createReq(double val, bool deg)
        {
            return new TrigRequest
            {
                Value = val,
                Unit = (Unit)(deg ? 0 : 1)
            };
        }
```

Appendix F. TrigCalculator Class: Trigonometric Remote Functions

```
        // Sine function which returns an instance of response struct
        0 references
        public Response Sine(double val, bool deg)
        {
            // Create a request object containing value and angle unit
            TrigRequest request = createReq(val, deg);

            // try communicate with the server
            try {
                // send request to grpc server
                MathResponse response = client.sine(request);
                // return a valid response object with the server response
                return new Response(true, response.Answer.ToString());
            }
            catch (RpcException e) { // if there is an RpcException
                // return an invalid response object with the error message
                return new Response(false, e.ToString());
            }
        }

        // Cosine function which returns an instance of response struct
        0 references
        public Response Cos(double val, bool deg)
        {
            // Create a request object containing value and angle unit
            TrigRequest request = createReq(val, deg);

            // try communicate with the server
            try {
                // send request to grpc server
                MathResponse response = client.cosine(request);
                // return a valid response object with the server response
                return new Response(true, response.Answer.ToString());
            }
            catch (RpcException e) { // if there is an RpcException
                // return an invalid response object with the error message
                return new Response(false, e.ToString());
            }
        }

        // Tan function which returns an instance of response struct
        0 references
        public Response Tan(double val, bool deg)
        {
            // Create a request object containing value and angle unit
            TrigRequest request = createReq(val, deg);

            // try communicate with the server
            try {
                // send request to grpc server
                MathResponse response = client.tan(request);
                // return a valid response object with the server response
                return new Response(true, response.Answer.ToString());
            } catch (RpcException e) { // if there is an RpcException
                // return an invalid response object with the error message
                return new Response(false, e.ToString());
            }
        }

    }
}
```

Appendix G. Calculator GUI Application: Setup

```
using System;
using System.Windows.Forms;
using CalculatorClient;

// using CalculatorGRPC namespace
namespace CalculatorGRPC
{
    // Windows forms object
    public partial class gRPCalculatorForm : Form
    {
        // variable initialisation
        private double runningTotal = 0;
        private string previousOp = "+";
        private bool firstFlag = true;
        private bool rpcError = false;
        private bool deg = true;

        // instantiating the TrigCalcualtor class from the CalculatorClient DLL
        private TrigCalculator grpcCalc = new TrigCalculator();

        // Form Constructor
        public gRPCalculatorForm()
        {
            InitializeComponent();
            grpcCalc.CreateChannel(); // create a gRPC channel on Form construction
        }

        private async void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            // close gRPC channel on Form closing
            await grpcCalc.CloseChannel();
        }
```

Appendix H. Calculator GUI Application: Helper Functions

```
// Number Input handler
3 references
private void numberInput(string data)
{
    // if number is first input after operation or startup
    if (firstFlag)
    {
        textBox1.Text = ""; // clear screen
        firstFlag = false;  // set firstFlag false
    }

    // if the last input was an rpcError
    if (rpcError)
    {
        // clear the flag and continue as normal
        rpcError = false;
    }
    // add data to textbox
    textBox1.Text += data;
}

// Maths Function Input handler
1 reference
private void functionInput(string data)
{
    // set temporary variable to 0
    double temp = 0;

    // if rpcError reset flag and leave temp set to 0
    if (rpcError) { rpcError = false; }
    // else set temp to value in text box
    else { temp = Convert.ToDouble(textBox1.Text); }

    // switch statement to handle various math functions and update the running total
    // The switch statement carries out the previously entered operation
    switch (previousOp)
    {
        case "+":
            runningTotal += temp;
            break;
        case "-":
            runningTotal -= temp;
            break;
        case "x":
            runningTotal *= temp;
            break;
        case "÷":
            runningTotal /= temp;
            break;
        case "√x":
            runningTotal = Math.Sqrt(temp);
            break;
        case "%":
            runningTotal = runningTotal/100*temp;
            break;
        case "xʸ":
            runningTotal = Math.Pow(runningTotal, temp);
            break;
    }
    // the current operation is then saved to be executed on the next function input
    previousOp = data;
    // add the new running total to the text box
    textBox1.Text = Convert.ToString(runningTotal);
    firstFlag = true; // set first flag for new number inputs
}

// helper function to handle the gRPC repsonse objects
3 references
private void handleTrigResponse(Response res)
{
    // if the result is valid
    if (res.valid)
    {
        // set the running total equal to the answer
        runningTotal = Convert.ToDouble(res.ans);
        // add the answer to the text box
        textBox1.Text = res.ans;
        // set previous operation to = which passes the switch statement without doing anything
        previousOp = "=";
    }
    else // else if result invalid
    {
        // write error message to text box
        textBox1.Text = "An error occured communicating with the server";
        // print full error to the console for testing
        Console.WriteLine(res.ans);
        // set rpcError flag to true
        rpcError = true;
    }
    // set first flag for new number inputs
    firstFlag = true;
}
```

Appendix I. Local Function Button Callbacks

```
// number button handler
10 references
private void numberBtn_Click(object sender, EventArgs e)
{
    // pass the button text to numberInput function
    numberInput(((Button)sender).Text);
}

// function button handler
8 references
private void functionBtn_Click(object sender, EventArgs e)
{
    // pass the button text to the functionInput function
    functionInput(((Button)sender).Text);
}

// clear button handler
1 reference
private void clrBtn_Click(object sender, EventArgs e)
{
    runningTotal = 0; // reset running total
    // display new running total
    textBox1.Text = Convert.ToString(runningTotal);
    previousOp = "+"; // reset previous op to plus
    firstFlag = true; // set first flag
    rpcError = false; // clear rpcError flag
}

// pi button handler
1 reference
private void piBtn_Click(object sender, EventArgs e)
{
    // send pi from math lib to numberInput
    numberInput(Convert.ToString(Math.PI));
}

// decimal point button handler
1 reference
private void decPtBtn_Click(object sender, EventArgs e)
{
    // if theres not already a decimal in the current input
    if (!textBox1.Text.Contains("."))
    {
        // send a decimal point to number input function
        numberInput(((Button)sender).Text);
    }
}

// angle unit button handler
1 reference
private void angUnitBtn_Click(object sender, EventArgs e)
{
    // get ref to button
    Button self = (Button)sender;
    // if degree flag set
    if (deg)
    {
        // update angle button to indicate radians
        self.Text = "Mode: rad"
        deg = false; // clear degree flag
    }
    else // if degree flag unset
    {
        // update angle button to indicate degrees
        self.Text = "Mode: deg";
        deg = true; // set degree flag
    }
}
```

Appendix J. Remote Trigonometric Function Button Callbacks

```
        // Trig button handlers
        // ===================

        // sine button handler
        1 reference
        private void sinBtn_Click(object sender, EventArgs e)
        {
            // call gRPC calculator DLL sine function, passing text box value and degree flag
            Response res = grpcCalc.Sine(val: Convert.ToDouble(textBox1.Text), deg: deg);

            // send response to handleTrigResponse function
            handleTrigResponse(res);
        }

        // cosine button handler
        1 reference
        private void cosBtn_Click(object sender, EventArgs e)
        {
            // call gRPC calculator DLL cosine function, passing textbox value and degree flag
            Response res = grpcCalc.Cos(val: Convert.ToDouble(textBox1.Text), deg: deg);

            // send response to handleTrigResponse function
            handleTrigResponse(res);
        }

        private void tanBtn_Click(object sender, EventArgs e)
        {
            // call gRPC calculator DLL tan function, passing textbox value and degree flag
            Response res = grpcCalc.Tan(val: Convert.ToDouble(textBox1.Text), deg: deg);

            // send response to handleTrigResponse function
            handleTrigResponse(res);
        }
    }
}
```