CMSC 417 Computer Networks		**Spring 2017**
Programming Assignment 4		
Assigned: March 8	*Due: March 17, 11:59:59 PM.*	*Weight: 5x*

1 Introduction

In this project, you will implement a remote procedure call (RPC) mechanism. RPCs are function calls that can be invoked on remote hosts (not the local machine). Consider hosts C and S: host C can invoke a RPC at S. C sends the function name and parameters to S, S computes the function locally using the parameters supplied by C and transmits the return value back to C.

You will then use this implementation to build a simple client and server, where the server performs addition operations for the client. You will make use of Google Protocol Buffers (GPB) [1] to serialize and deserialize the structured data sent as part of the RPC protocol (e.g., function arguments). Serialization is the process of converting a structured data object into a stream of bytes that then may be stored or transferred. Deserialization is the reverse process.

2 Protocol

The RPC protocol has two message types: *Call* and *Return*. Both of these message types are specified in the Google Protocol Buffer (GPB) language [2], which you should read and understand. The message types are specified as follows:

```
message Call {
  required string name = 1;
  required bytes args = 2;
}

message Return {
  required bool success = 1;
  required bytes value = 2;
}
```

The *Call* message consists of two required fields: the 'name' of the function as a string, and the 'args' (arguments) of the function as an array of bytes. The *Return* message consists of two required fields: the 'success' of the RPC as a boolean, and the 'value' returned by the function as an array of bytes. In these cases, the byte arrays will hold serialized content specific to a given function.

The client begins the RPC protocol by sending a *Call* message to the server. The server then deserializes the message and reads the 'name' field. The server maintains a mapping of registered names that map to specific functions, each of which will handle the following steps:

1. Deserialize the 'args' field (which was passed in as an argument) according to the specific GPB message type that is used for this function's arguments.

2. *Call* the actual function on the server with the deserialized arguments and note the return value.

3. Serialize the return value according to the specific GPB message type that is used for this function's return value.

At this point, the server will place the serialized return value in a *Return* message, and send that back to the client.

The server will support two functions: 'add' and 'getAddTotal'. The 'add' function will take two 32-bit integer arguments and return their sum as a 32-bit integer. The 'getAddTotal' function will return the running total of all values added together by the 'add' function for a given client.

3 GPB Example

In this section, we will work through an example that demonstrates how to use GPB to implement the required RPC functionality. The code used in this example can be found in the 'assignment4' directory in the 'materials' repository. You should read through the code 'rpc.c', the protocol buffer message specification 'rpc.proto', and the 'Makefile'.

Let us consider a very basic function 'invert' that is implemented by the server as follows:

```
bool invert(bool v) {
  return !v;
}
```

The GPB message types used for the arguments and return value, which are serialized and placed the *Call* and *Return* messages, are as follows:

```
message InvertArguments {
  required bool v = 1;
}

message InvertReturnValue {
  required bool notv = 1;
}
```

The server should maintain a mapping from the 'invert' name to a function called 'invertWrapper'. The 'invertWrapper' function takes in an array of bytes, which it will deserialize using the InvertArguments GPB message and obtain the value of v. The 'invertWrapper' function will then call invert(v), obtain the return value from the function, and serialize it using the InvertReturnValue GPB message.

4 Client

The client will be a command line utility which takes the following arguments:

1. **-a <String>** = The IP address of the machine that the server is running on. Represented as a ASCII string (e.g., 128.8.126.63). Must be specified.
2. **-p <Number>** = The port that the server is bound to and listening on. Represented as a base-10 integer. Must be specified.
3. **-t <Number>** = The time in seconds between 'add' RPCs. Represented as a base-10 integer. Must be specified, with a value in the range of [0,30]. A value of 0 means that the next RPC should execute immediately after the previous RPC returns.

4. **-n <Number>** = The number of 'add' RPCs between 'getAddTotal' RPCs. Must be specified, with a value in the range of [1,10].

An example usage is:

client -a 128.8.126.63 -p 41714 -t 5 -n 10

The client will first connect to the server based on the values of (-a) and (-p). Afterwards, client will perform 'add' RPCs every (-t) seconds, drawing the two values to supply as arguments uniformly at random from [0,1000]. The client will print a single line after the RPC completes that contains the two values being added, as well as the result. These values must be printed in base-10, and must be separated by a space.

Every (-n) 'add' RPCs, the client will additionally perform a 'getAddTotal' RPC immediately after the previous 'add' RPC. The client will print a single line that contains the summation total as returned by the server, with the value printed in base-10.

5 Server

The server will be a command line utility which takes the following arguments:

1. **-p <Number>** = The port that the server will bind to and listen on. Represented as a base-10 integer. Must be specified.

An example usage is:

server -p 41714

The server will open a TCP socket, and bind and listen on the port specified by (-p). The server must be able to handle many concurrent clients via poll or select. The server will support the 'add' and 'getAddTotal' RPCs as specified in Section 2.

6 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.

2. The directory for your project must be called 'assignment4' and be located at the root of your Git repository.

3. You must provide a Makefile that is included along with the code that you commit. We will run 'make' inside the 'assignment4' directory, which must produce 'client' and 'server' executables also located in the 'assignment4' directory.

4. You must submit code that compiles in the provided VM, otherwise your assignment will not be graded.

5. Your code must be -Wall clean on gcc/g++ in the provided VM, otherwise your assignment will not be graded. Do not ask the TA for help on (or post to the forum) code that is not -Wall clean, unless getting rid of the warning is the actual problem.

6. You are not allowed to work in teams or to copy code from any source.

References

[1] Google Protocol Buffers. https://developers.google.com/protocol-buffers.

[2] Google Protocol Buffers: Language Guide. https://developers.google.com/protocol-buffers/docs/proto.