CMSC 417 Computer Networks **Fall 2017**

Programming Assignment 4

Assigned: October 13 *Due: October 23, 11:59:59 PM.* *Weight: 1x*

1 Introduction

In this project, you will use Google Protocol Buffers (Protobufs) to implement a Remote Procedure Call (RPC) mechanism.

RPC is a generic term for a function call that is serviced in a different address space than the caller's. RPC necessarily entails a protocol for the caller's process, C, to encode the function name and arguments and send them to the servicing process, S, as well as for S to encode the return values and send them to C. In this assignment, the encoding will use *serialization*, a term for converting a language level data object to a sequence of bytes. The inverse operation – reconstituing a data object from a sequence of bytes (as stored on disk or sent over the network) – is called deserialization. Protobufs is precisely a framework for describing data objects and serializing/deserializing them.

2 RPC Protocol

For this assignment, the RPC protocol has two messages: (1) `Call`, which encapsulates the function invocation, and (2) `Return`, which encapsulates the function return. In Protobufs notation, the message types are specified as:

```
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.

The `args` and `value` byte arrays hold serialized content specific to a given function. Specifically, you must define additional Protobuf messages to describe each function's argument list and return value (`void` argument lists and return values aside). The serialization of the Protobuf message corresponding to a function's argument list is the value for `Call`'s `args` field; the serialization of the Protobuf message corresponding to a function's return value is the value for `Returns`'s `value` field.

Logically, an RPC client is a connection to an RPC server. Over the client's lifetime, the client may make multiple RPCs over the same connection. To wit, if a process connects to an RPC

server, issues an RPC, disconnects, and then re-connects to the RPC server, the server considers the second connection a new client.

To aid in receiving messages, you will likely want to prepend a simple header to the serialization of `Call` and `Return` that denotes their length.

3 RPCs

You must implement two functions as RPCs: `add` and `gettotal`. Implemented locally, the functions are:

```
static uin32_t total = 0;

uint32_t
add(uint32_t a, uint32_t b)
{
    total += (a + b);
    return (a + b);
}

uin32_t
gettotal(void)
{
    return (total);
}
```

Note that:

- Each RPC client has its own version of the variable `total`; `total` is the accumulator of the client's `add` operands.
- Both RPCs are blocking; the caller waits for the RPC to return before calling another function, whether local or remote.
- For this assignment, you do not need to worry about integer overflow.

4 Setup

You may user either the C++ or the C version of Protobufs. Google officially maintains a C++ version of Protobufs, which may be installed by:

```
sudo apt-get install libprotobuf-dev
sudo apt-get install protobuf-compiler
```

The C version of Protobufs is maintained outside of Google, and may be installed by:

```
sudo apt-get install libprotobuf-c-dev
sudo apt-get install protobuf-c-compiler
```

Please read the Protobufs language guide [1]. There is a tutorial for the C++ version [3] and for the C version [2]. Additionally, the `assignment4` directory of the `materials` repo contains an example that uses the C version to create an RPC for the function:

```
bool
inverse(bool b)
{
    return (!b);
}
```

5 Driver Programs

5.1 Client

The client is a command line utility that takes the following arguments:

1. **-a <String>** = The IP address of the machine that the server is running on. Represented as an 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 `gettotal` 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 connects to the server based on the values of (-a) and (-p). The client then performs `add` RPCs every (-t) seconds, drawing the two values to supply as arguments uniformly at random from [0,1000]. The client must 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 additionally performs a `gettotal` RPC. The client then prints a single line that contains the total as returned by the server, in base-10.

The client should run until terminated.

5.2 Server

The server is a command line utility that takes the following arguments:

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

An example usage is:

server -p 41714

The server opens a TCP socket, and binds and listens on the port specified by (-p). The server must be able to handle many concurrent clients via poll or select. The server must support the `add` and `gettotal` RPCs as specified in Section 2.

6 Grading

Since the on-the-wire protocol is intentionally incomplete, we will only test your client with your server, and vice-versa.

7 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: Language Guide (proto2). `https://developers.google.com/protocol-buffers/docs/proto`.

[2] protobuf-c Example. `https://github.com/protobuf-c/protobuf-c/wiki/Examples`.

[3] Protocol Buffer Basics: C++. `https://developers.google.com/protocol-buffers/docs/cpptutorial`.