RPC Lab

Machines in a distributed system must communicate with one another. One method of doing this is using Remote Procedure Calls (RPCs), which make communication between machines look like ordinary function calls. Effectively, RPCs abstract away the underlying serialization required for raw messaging and allow for an interaction like this:

- A client calls `remoteFileSystem->Read("rutabaga")`.
- The underlying RPC library serializes this function call as a request and sends it to the server.
- The RPC library deserializes the request on the server and runs `fileSys->Read("rutabaga")`, which returns a response.
- The RPC library serializes the response, sends it to the client, and deserializes it to give the client the return value for its function call.

Effectively, this allows the client to call a function on an entirely different machine as if it was just calling any other function.

> **SERIALIZATION**
> Serialization is the process of converting structures into bytes that can be sent via communication protocols like TCP, while deserialization recovers the original structures from the serialized bytes. Serialization formats specify how serialization and deserialization should take place, and allow communicating processes to understand the bytes being received from one another. For example, JSON and YAML are two common serialization formats that often find use in configuration files.

In this lab, you will be familiarizing yourself with `rpcgen`, a protocol compiler that helps with writing RPC applications in C. You will do so by implementing a very basic key/value (KV) store, which allows clients to put keys in a server-side hash map and read them out later by making requests to the server.

For a rundown of how `rpcgen` works, please read this guide. However, we will be covering all of the content relevant to the upcoming MapReduce homework in this lab.

Getting started

To get started, log in to your development environment and get the starter code.

```
cd ~/code/personal
git pull staff main
cd lab-rpc
```

You will need to install `rpcbind`:

```
sudo apt-get install rpcbind
```

Usage

After you have installed `rpcbind`, you will need to run `sudo service rpcbind start`, this will allow the Docker workspace to start running the `rpcbind` calls. **You will need to run this command once everytime you restart your Docker workspace.**

To compile the code, run `make`. This will generate the necessary stubs using `rpcgen` as well as the `server` and `client` binaries in the `bin/` directory. To test the binaries, follow the steps below:

1. Start the `server` binary with no arguments by running `./bin/server` from the root directory of the homework.
2. In another terminal, run the client with the appropriate subcommand:

   ```
   client subcommands:
        example Make an EXAMPLE RPC
        echo    Make an ECHO RPC
        put     Make a PUT RPC
        get     Make a GET RPC
   ```

   Currently, only the EXAMPLE RPC is implemented (it simply adds 1 to the provided input), but usage details for all of the RPCs that you will be implementing are shown below:

   ```
   client example [INPUT]
   client echo [MSG]
   client put [KEY] [VALUE]
   client get [KEY]
   ```

   For now, you can try running `./bin/client example 1`. You should see an output of `2`.

NOTE

We strongly recommend using `tmux` to multiplex a single terminal instead of opening separate terminals for the client and server.
