


Project 8

Due May 3 by 11:59pm **Points** 100

In this project you will be implementing a simplified version of a classic paper, [Implementing Remote Procedure Calls](https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf)  (<https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf>)

Administrivia

- **Due Date** by May 3rd, 2023 at 11:59 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: This is a group assignment. It can be solved individually, or in a group of 2 students. We highly recommend doing this in a group of 2. **[IMPORTANT] Complete the canvas [project 8 quiz](https://canvas.wisc.edu/courses/330415/quizzes/440511) (<https://canvas.wisc.edu/courses/330415/quizzes/440511>) to tell us about your group.**
- Copying code (from others) is considered cheating. [Read this](<http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html>) (<http://pages.cs.wisc.edu/~remzi/Classes/537/Spring2018/dontcheat.html>) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- Some tests *will be* provided at `~cs537-1/tests/P8`. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P8/README` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Copy all your files to `~cs537-1/handin/login/P8` where login is your CS login.
- Starter code can be found at: `/home/cs537-1/tests/P8/starter_code`
- **This project is for extra credit**

Remote Procedure Calls

Remote Procedure Calls (RPC) is a very popular way to simplify programming distributed systems. They allow for non-experts to be able to easily execute functions on a remote server without having to worry about the networking, scheduling, or management of these tasks. They also give very powerful at-most once semantics, meaning that if you call this function it will run at most one time on the server. The client should also be able to notice that the function has not executed, it just may not be able to resolve this. Under normal circumstances these functions will run exactly once. Note that this is different from the exactly once semantics of running a normal function.


One major limitation of these functions is that, because you are running remotely, it requires that all data be passed by value. Practically this limits how much data can actually be transferred per function

call.

The main goals of this assignment are to:

- Familiarize yourself with basic distributed systems
- Learn how to have multiple processes interact
- Learn how to use pthreads for logical separation of functions rather than parallelizing

Background

To be able to complete this assignment you will need to understand the basics of thread creation and detaching threads. It is also important to understand the RPC protocol described in [Implementing Remote Procedure Calls](https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf)  (<https://web.eecs.umich.edu/~mosharaf/Readings/RPC.pdf>). While also not required, it is helpful to understand very basic networking protocols, however we will be giving you all of the networking code.

The following is also described in the following book chapters:

- [Intro To Threads] (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf> (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf>))
- [Common Concurrency Problems] (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf> (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf>))
- [Event-based Concurrency] (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf> (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf>))
- [Distributed Systems] (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf> (<https://pages.cs.wisc.edu/~remzi/OSTEP/threads-events.pdf>))
- [Network File Systems] (<https://pages.cs.wisc.edu/~remzi/OSTEP/dist-nfs.pdf> (<https://pages.cs.wisc.edu/~remzi/OSTEP/dist-nfs.pdf>))

Project Specification

Your RPC implementation will implement a client library that can be integrated into a client application, and a server that can handle client library requests.

A client will send the server a message containing a client id as well as a sequence number. The client id is uniquely randomly selected on startup (rand() is okay). The sequence number is a monotonically increasing number that is used to track if requests are duplicate. The client needs to send messages to the server and block until either the message times out or the it receives a message from the server. In

the case of a timeout, the client will try sending the message to the server again. If the client receives no responses after 5 attempts, the client should exit with an error message.

The server will track clients that have connected to the server. When a client requests to have the server run a function it will execute the chosen function if the server has a definition for the function. To prevent duplicate requests and enforce the at-most once semantics, if it receives a message that is less than the current tracked sequence number for that client it will simply discard it. If it receives a message that is equal to the sequence number it will reply with either the old value that was returned from the function, or a message indicating that it is working on the current requests. If it receives a message with a sequence number that is greater than the current tracked sequence number this indicates a new request and it will start a thread to run this task that will reply independently to the client.

The RPC server should take one command line argument:

- **port:** The port that the server will bind itself to

`server_functions.h` and `server_functions.c` contain the implementations of the RPC stubs for the server to execute. If you use the given Makefile this will build all of the file with your `server.c` file automatically. This file contains the following implementations:

- `idle`
 - This sleeps the thread for a given number of seconds. In practice this is just a wrapper for sleep.
- `get`
 - This reads a value from a server local int array. If it is out of bounds it returns -1.
- `put`
 - This sets a value from a server local int array. If it is out of bounds it returns -1.

Otherwise the RPC server implementation is completely up to you. It will be helpful for you to look and use `udp.h` and `udp.c`. This is the easiest way for you to receive and send communications to your clients. There is more information about this in the hints. It will also be helpful to understand the call table from the paper as well as you will likely implement something very similar in your implementation. Without duplicating the paper, the call table can be summarized as an array of structs that contains a unique identifier for you client, the last sequence number for that client, and the last result for that client. This table can then be searched to find the client context and implement the sequence number handling reiterated below:

- message arrives with sequence number i :
 - $i > \text{last}$: new request - execute RPC and update call table entry
 - $i = \text{last}$: duplicate of last RPC or duplicate of in progress RPC. Either resend result or send acknowledgement that RPC is being worked on.
 - $i < \text{last}$: old RPC, discard message and do not reply

This may raise the question of what happens if the client or server crashes? The client will get a new `client_id` and the client will need to restart their requests. From a client perspective we will be unsure if the last message completed, but this is okay as it still fulfills our at-least once guarantee. On server crash we will lose the entire Call Table. In the paper there are solutions to this, but in this project we are not going to handle this scenario and we can assume that all call state is lost.

The server is required to have the following functionality:

- Track up to 100 connected clients
- Execute idle, get, and put commands (given in the starter code)
- Receiving packets from a client handling the execution
 - Execute the requested function if the sequence number is greater than what the server has tracked for the client. This should also begin running a new thread that will allow for multiple clients to connect at the same time
 - Resend results from the most recent RPC for every client if the sequence number is equal
 - Reply with an acknowledgement (ACK) for in progress requests
 - Ignore requests that are older than the most recent requests sequence number

The client library will not have a main function and will integrate into provided test programs. The starter code for the client library (in `client.h`) will contain the interface that needs to be implemented. These are the only functions that need to be implemented in `client.c`.

- `RPC_init`
 - This function will initialize the `rpc_connection` struct and do any other work you need to do to initialize your RPC connection with the server
- `RPC_idle`
 - This function will initiate and block until the idle RPC is completed on the server
- `RPC_get`
 - This function will initiate and block until the get RPC is completed on the server
- `RPC_put`
 - This function will initiate and block until the put RPC is completed on the server
- `RPC_close`
 - This function will do any cleanup that you need to do for your RPC variables

The client library should have the following functionality:

- Sending idle, get, or put requests to the RPC server and blocking until response from the server or throwing an exception on no response.
- Retrying RPC requests on a chosen (short) timeout interval up to 5 times
- Ignore packets that are for other client ids, or to old sequence numbers
- Delaying retrys for 1 second on receiving an ACK from the server

Messages between the server and client have no requirements, however there are a few things that will likely need to contain in them:

- sequence numbers: This is what you will use to determine if this is a rebroadcast of an old request, or a new request from the client
- client_id: This is needed for the server to determine which call table entry to associate with the client, and for the client to determine if this message is actually intended for it.

Grading & Submission Details

You will write your code in `server.c` and `client.c`, the Makefile (to allow you to modify it), and any other files you need for your implementation.

It is very important that you DO NOT change the headers. These will be used to integrate into our tests and if these are changed we will likely not be able to run your code.

Your code will be first measured for basic functionality under ideal conditions and then tested under less than ideal conditions. We will do this to verify properties like at-most once semantics, and durability under unstable network conditions.

For submission make sure to include `server.c` and `client.c`.

Hints and Considerations

While this project is open-ended in terms of implementation, here are a few considerations best taken into account while you code.

- you can format your message payload carefully to avoid having to implement separate handlers for every type of message. This can save you time in your implementation and significantly simplify your code. One suggested format for doing so, is to have an identifier at the start of your struct that lets the user know what kind of message is being sent. This can then be used to decide which type to cast the payload to.

- Sometimes the port that you are using can not be freed on a segmentation fault. It is smart to be able to easily change the port that the client and server are using so you can keep working while these ports are occupied. They will eventually be freed on their own.
- If you are writing structs (or technically code) that is being shared between the client library and the server, you can make a header file that is included in both to reduce code duplication without changing the Makefile.
- Summary of the `udp.h` and `udp.c` functions:
 - `init_socket` is meant to initialize a socket for receiving UDP packets. You simply pass it the port you want to listen on, and it will return you a socket you can pass to other functions to receive packets on.
 - `receive_packet` is meant to receive a packet listening on a socket created from `init_socket`. This will wait until a packet is received and fill in packet info. This is useful for the server side.
 - `receive_packet_timeout` is very similar to `receive_packet`, but the main difference is that it takes an extra timeout variable. This function works identically, however if the socket times out, it will return a packet with a length of less than 0.
 - `send_packet` will send a packet to a target destination. If you do not already have a target destination (possibly from receiving a packet) you will need to call `populate_sockaddr` to generate the destination to fill in the target and `slen` parameters.
 - `populate_sockaddr` fills in the values for a UDP `sockaddr` variable. This will be useful for building the server destination address on the client side. To use `populate_sockaddr` the following is the intended usage:

```
// init tx connection
struct sockaddr_storage addr;
socklen_t addrlen;
populate_sockaddr(AF_INET, dst_port, dst_addr, &addr, &addrlen);
rpc.dst_addr = *((struct sockaddr *)&addr);
rpc.dst_len = addrlen;

send_packet(rpc->recv_socket, rpc->dst_addr, rpc->dst_len, (char *)message, message->message_size);
```

- `close_socket` is a very simple function that will close and clean up a socket variable.
- It is possible that the port you are trying to use is used by another student on the CSL machines. In this case you may have to change the port that the test cases run on. You can then start the server on a different port using: `./server <port>`
- If you are remotely connected to the CSL machines you will once again need to make sure that you are running all of these processes on the same device. This means connecting to the machine name rather than `best-linux`.
 - Another option is to run the server process in the background using `./server <port> &` and then running the client. If you are trying to debug the server you can then kill the process using `kill server`, or by bringing the pid to the foreground and terminating it there.