CS454 - Assignment 3

Mitchell Grenier - 20438709, Dan Li - 20383841

July 2015

This is the formal documentation for RPCLib, our non compiler distributed function calling mechanism.

Building
For our project we went with as structured approach to the source code as we could. For this reason, our building steps are a little different than most I would gather. The first step is to compile the library and binder executable with dependencies. This is done by executing:

make release

This will compile the library, the binder and then copy the rpc.h file into the release folder If you wish to use our server and client tests (of which there are 14), you would use

make library
make binder
make server
make client

This will compile all the executables into the build folder and they can be executed from there.

Notes on building
For the purposes of understanding what is happening and marking, before any compilations, setting VERBOSE_OUTPUT to 1 will enable many debugging output lines useful for understanding execution flow and steps taken.

We also use clang++ for compilation and testing so this should also be during your use of this as well. Our make file is quite extensive on the flags required for compilation of clients and servers against the library and can be consulted as a guide if it is at all unclear. We have also tested copying our entire release folder to another folder and making the server and client against this (the binder is made automatically with make release). These are the commands we used to compile and like without the aid our our makefile (this was done to ensure we were not accidentally relying on other external headers/files or other constructs).

make release
cp release/* $TESTING_DIR
cp client/* $TESTING_DIR
cp server/* $TESTING_DIR
cd $TESTING_DIR
clang++ -c server_functions.c client1.c server_function_skels.c server.c
clang++ -pthread -L. server.o server_function_skels.o server_functions.o -o server -lrpc
clang++ -pthread -L. client1.o -o client -lrpc

Data Messages and Marshalling
Our system messages are packed quite tightly, with no extraneous bytes if it can be helped. Most messages have unique formats for messaging specific to their duty.

RPC Register
0x0 - Message identifier signifying that this is an RPC Register call
4 bytes - Int - The port that this function is available on.
4 bytes - Int - The length of the function name.
n bytes - String - The function name.
4 bytes - Int - The number of parameters that this function takes.
m*4 bytes - Ints - The int descriptors for each parameter.

RPC Call (Client/Binder)
0x1 - Message identifier signifying that this is an RPC Register call
4 bytes - Int - The length of the function name.
n bytes - String - The function name.
4 bytes - Int - The number of parameters that this function takes.
m*4 bytes - Ints - The int descriptors for each parameter.

RPC Call (Client/Server)
0x1 - Message identifier signifying that this is an RPC Register call
4 bytes - Int - The length of the function name.
n bytes - String - The function name.
4 bytes - Int - The number of parameters that this function takes.
m*4 bytes - Ints - The int descriptors for each parameter.
k bytes - Packed data - The data packed tightly together (no delimiters) that make up all sent data. For example, if the function takes 3 ints, there will be 12 bytes here. If the function takes an array of characters and then 2 ints, we will have all the bytes that make up the string (without a null terminator) followed by 8 bytes (4 for each int).

RPC Terminate
0x2 - If received by binder, forwards to all servers then terminates, if received by server, terminate.

Functional Databases and Round Robin

Binder:

The binder maintains a deque of all servers that have registered functions. This is a deque of structures that contains a list of all available functions for that server. When a client requests a function from the binder, the servers are checked in the order they are supposed to execute until a server is found (if one is not found, an error is returned to the client). When the function is found, that server and port will be sent to the client in the same format as RPC call but without parameter info and is then moved to the back of the deque. Overloading is handled when a register call comes in. We will search though the functions provided by that server by comparing all params (ignoring array lengths), and the name. If we find that function already there, we will tell the client that with a two byte message (REGISTER_FAIL, REGISTER_EXISTS). Otherwise we will either return a welcome (REGISTER_SUCCESS, REGISTER_WELCOME) if this is the first function they've registered, or we will return (REGISTER_SUCCESS, REGISTER_F_ADD).

Server:

When the server calls RPC Register and needs to store the skeleton function. We decided to store it in a map. The key for that function will be generated by taking the ArgTypes and changing the length of every arg that are greater than or equal 1 to be 1. This insures that when RPC registers a function that is different from client's input, the client will still be able to pass it its arguments to the skeleton. After the server saves this skeleton in the map, it will wait until the client connects to itself and pass its arguments. The server will then do a lookup using the client's argTypes, again setting those whose length is greater than or equal 1 to be 1.

Execution Procedure

The RPC_EXECUTE acts like a server to process incoming client requests. The first thing it does is that it calls listen(sd,5) where sd is the server descriptor and similar to what i have done in assignment2, i created a FD_Master to store the connections and use select to multiplex.

When a new connection comes i, I check the call_type, if it is a terminate. I close the server and set FD_MASTER to zeros. If the call is from RPC_CALL. I get the argTypes and args, then put these in a structure , which will be passed as arguments in a pthread function. The pthread function is responsible for running the skeleton function and sending the response.

Termination Procedure

The termination procedure is outlined in the message specification for termination. To reiterate, it consists of a single byte, which is sent from a client to the binder. The binder will then forward that to all servers then terminate itself. The servers will verify that message is from the binder before terminating.

Error Codes

Our code has no unique error codes that need to be handled by server or client programs. -1 is the only important error code, and is used when an RPC Call fails for some reason. Other errors if serious are printed to standard out, such as registering a function in a server.

Implementation Gaps and Extra Functionality
As far as we are aware everything in the spec that needs to be there is there. We did not attempt the bonus so rpcCacheCall will always return -1.

If you look at our submitted server implementation you can see the extensive tests we ran to test our program (including copying files from client to server via f12. These are all fully functional and do exactly as their names suggest. I would however say perhaps the only one that qualifies as extra functionality, is f12 which can copy a file, to a location on the server.