CS454/654 Assignment 3

Due: 14 March 2018 (4:00pm)
Returned: 5 April 2018
Appeal deadline: 12 April 2018
See course webpage for late submission policy

See course webpage for TA contact/office hours

You are required to do this assignment with a partner who is also a CS454/654 student this term. You are responsible for finding your partners. Both members of a team will receive the same mark. Both CS454 and CS654 students are to do the assignment.

For this assignment you are required to implement a crude version of Remote Procedure Call. Normally this would require some degree of compiler support. However, we will simplify the interface so that the basic RPC is there, and it is only a matter of syntactic sugar to clean up appearances.

For the RPC implementation, we require three processes: a client, a server, and a binder. You will implement, using TCP/Sockets, the RPC library including a number of functions described later, and the binder. It is up to the user (TA in this case) to write the client, the main server program, the server function implementations, and the server function stubs using your library where applicable. You may assume that they are (almost) perfect coders (this assumption is there so that you don't have to do a ton of code checking for correctness!). The mistakes they may make, and which you have to check for, are things like not registering before invoking the server execute function, calling a RPC that isn't there yet, starting the client before the server, etc. They never make mistakes with respect to parameters in the function calls.

The system can be briefly divided into the following three portions:

1. The client requests from the binder the server identifier (IP address or hostname) and port number of a server capable of handling the request. The client side marshals the parameters into a request message, sends the request to the server, retrieves the result and returns.

2. The server creates a connection socket that listens for client requests and, on receiving a request, identifies the desired server procedure, calls it with the appropriate parameters (extracted from the client request), and returns the results to the client. The server registers the server procedures with the binder and keeps a seperate TCP connection to the binder open so the binder knows it is still up.

3. The binder takes registration requests from server processes and maintains a database of servers and associated procedures. It also services location requests from client processes, either returning the server identifier (IP address or hostname) and port information for a suitable server or indicating that no such server exists. Finally, since we may wish to terminate the entire system in a reasonably graceful fashion, the binder also handles terminate-request

messages causing it to exit and, indirectly, also causing all servers to terminate. Clients can be expected to terminate themselves gracefully without assistance.

The following are detailed specifications.

1 Client Side

The client will execute a RPC by calling the `rpcCall` function. The signature of this function is:

```
int rpcCall(char * name, int * argTypes, void ** args);
```

First, note that the integer returned is the result of executing the `rpcCall` function, not the result of the procedure that the `rpcCall` was executing. That is, if the `rpcCall` failed (*e.g.* if there was no server that provided the desired procedure), that would be indicated by the integer result. For successful execution, the returned value should be 0. If you wish to indicate a warning, it should be a number greater than 0. A severe error (such as no available server) should be indicated by a number less than 0. The procedure that the `rpcCall` is executing is, therefore, not able to directly return a value. However, it may do so via some argument.

The `name` argument is the name of the remote procedure to be executed. A procedure of this name must have been registered with the binder.

The `argTypes` array specifies the types of the arguments, and whether the argument is an "input to", "output from", or "input to and output from" the server. Each argument has an integer to encode the type information. These will collectively form the `argTypes` array. Thus `argTypes[0]` specifies the type information for `args[0]`, and so forth.

The argument type integer will be broken down as follows. The first byte will specify the input/output nature of the argument. Specifically, if the first bit is set then the argument is input to the server. If the second bit is set the argument is output from the server. The remaining 6 bits of this byte are currently undefined and must be set to 0. The next byte contains argument type information. The types are the standard C types, excluding the null terminated string for simplicity.

```
#define ARG_CHAR    1
#define ARG_SHORT   2
#define ARG_INT     3
#define ARG_LONG    4
#define ARG_DOUBLE  5
#define ARG_FLOAT   6
```

In addition, we wish to be able to pass arrays to our remote procedure. The lower two bytes of the argument type integer will specify the length of the array. Arrays are limited to a length of 2^{16}. If the array size is 0, the argument is considered to be a scalar, not an array. Note that it is expected that the client programmer will have reserved sufficient space for any output arrays.

You may also find useful the definitions

```
#define ARG_INPUT      31
#define ARG_OUTPUT     30
```

For example, "`(1 << ARG_INPUT) | (ARG_INT << 16) | 20`" represents an array of 20 integers being sent to the server. "`(1 << ARG_INPUT)|(1 << ARG_OUTPUT)|(ARG_DOUBLE << 16)|30`" on the other hand is 30 doubles sent to and returned from the server.

Since we do not know how many arguments there are, the last value we pass in the `argTypes` array is 0, thus the size of `argTypes` is 1 greater than the size of `args` (please refer to the sample code given later). The `args` array is an array of pointers to the different arguments. For arrays, they are specified by pointers in C/C++. We can use these pointers directly, instead of the addresses of the pointers. For example, in the case of `char stringVar[] = "string"` we will use `stringVar` in the argument array, not `&stringVar`.

Thus, if the client wished to execute `result = sum(int vect[LENGTH])`, the code would be:

```
// result = sum(vector);
#define PARAMETER_COUNT 2       // Number of RPC arguments
#define LENGTH 23               // Vector length

int argTypes[PARAMETER_COUNT+1];
void **args = (void **)malloc(PARAMETER_COUNT  * sizeof(void *));

argTypes[0] = (1 << ARG_OUTPUT) | (ARG_INT << 16);           // result
argTypes[1] = (1 << ARG_INPUT)  | (ARG_INT << 16) | LENGTH;  // vector
argTypes[2] = 0;                                             // Terminator

args[0] = (void *)&result;
args[1] = (void *)vector;

rpcCall("sum", argTypes, args);
```

Note that the number of output arguments is arbitrary and they can be positioned anywhere within the `args` vector.

To implement the rpcCall function you will need to send a location request message to the binder to locate the server for the procedure. If this results in failure, the rpcCall should return a negative integer, otherwise, it should return zero. After a successful location request, you will need to send an execute-request message to the server. The specific message communication will be described in the Protocols description (Section 5).

2 Server Side

On the server side, there is a main server program, several server functions and function skeletons. Server functions provide actual services, such as `sum(int a, int b)`. For each server function, there is a corresponding skeleton that does the marshalling and unmarshalling for the actual server function.

The server first calls `rpcInit`, which does two things. First, it creates a connection socket to be used for accepting connections from clients. Secondly, it opens a connection to the binder, this connection is also used by the server for sending register requests to the binder and is left open as long as the server is up so that the binder knows the server is available. This set of permanently open connections to the binder (from all the servers) is somewhat unrealistic, but provides a straightforward mechanism for the binder to discover server termination. The signature of `rpcInit` is

```
int rpcInit(void);
```

The return value is 0 for success, negative if any part of the initialization sequence was unsuccessful (using different negative values for different error conditions would be a good idea).

The server then makes a sequence of calls to rpcRegister to register each server procedure. The signature of the register function is

```
int rpcRegister(char *name, int *argTypes, skeleton f);
```

where `skeleton` is defined as

```
typedef int (*skeleton)(int *, void **);
```

This function does two key things. It calls the binder, informing it that a server procedure with the indicated name and list of argument types is available at this server. The result returned is 0 for a successful registration, positive for a warning (e.g., this is the same as some previously registered procedure), or negative for failure (e.g., could not locate binder). The function also makes an entry in a local database, associating the server skeleton with the name and list of argument types. The first two parameters are the same as those for the `rpcCall` function. The third parameter is the address of the server skeleton, which corresponds to the server procedure that is being registered.

The skeleton function returns an integer to indicate if the server function call executes correctly or not. In the normal case, it will return zero. In case of an error it will return a negative value meaning that the server function execution failed (for example, wrong arguments). In this case, the RPC library at the server side should return an RPC failure message to the client.

The server finally calls rpcExecute, which will wait for and receive requests, forward them to skeletons, and send back the results. The `rpcExecute` function has the signature:

```
int rpcExecute( void )
```

It hands over control to the skeleton, which is expected to unmarshall the message, call the appropriate procedures as requested by the clients, and marshall the returns. Then `rpcExecute` sends the result back to the client. It returns 0 for normally requested termination (the binder has requested termination of the server) and negative otherwise (*e.g.* if there are no registered procedures to serve).

`rpcExecute` should be able to handle multiple requests from clients without blocking, so that a slow server function will not choke the whole server.

Multiple servers can run on the same machine (having a single IP address). You are recommended to use a dynamic port here to avoid any conflicts. We will only mark based on what we observe during the test. If hardcoded port numbers are used, you will lose marks. This applies to the binder port as well.

To implement the register function you will need to send a register message to the binder. The specific message communication will be described in Section 5 under Protocols description.

3 Binder

The binder accepts registration requests and location requests, generating replies as defined in the Protocols section (Section 5). It must maintain a database of procedures that have been registered with it, including arguments, so that when it receives a location request it can respond appropriately. The database will be of the form

```
procedure signature, location
```

The specific details of how you manage this database are up to you. Note that for arrays, the server cannot predict the exact length of the array for the input or output argument, when registering it with the binder. Therefore, you should disregard the array length while matching a client request to functions registered at the binder. An important thing to note here is that two functions with signatures only differing in array lengths (`> 0`) are to be considered the same, while they would be considered different only if one accepts a scalar and the other an array.

Your binder should handle function overloading. It is possible that a single server registers different functions with the same name and different arguments. Multiple servers can also register functions with the same name yet different arguments. You can assume that if different servers register functions with the same name and arguments, their respective skeletons will provide identical functionality. If the same server registers a function with the same name and arguments twice, you should override the previous entry and only retain information about the latest provided skeleton.

When the binder receives a request from a client that asks for a procedure provided by more than one server, the binder should return the servers according to a simplified round-robin algorithm. The requests should be rotated among servers capable of serving them turn by turn, each one getting an equal opportunity. No server should serve again until all other servers have also served a request, unless there is no other server that can serve the request. As an example, suppose server A has functions f(), g(), and h(). Server B has functions f() and g(). Then the requests series f, h, g, f presented by a single client C, will be processed by A, A, B, and A.

There must be some mechanism for the server and the client to know where the binder is and what port it is listening to. Since this will be dynamic and since we have no control over the /etc/services files, we will use two environment variables. Specifically, the binder must print two distinct lines of the form

```
BINDER_ADDRESS <machine>
BINDER_PORT    <port number>
```

where `<machine>` is the machine name or IP address where the binder is executing and `<port number>` is the port number that the binder is listening to. This allows the user at the server or client machine, before executing the server or client, to set these values in the shell. The server and client stubs must read these from the environment and call the binder appropriately.

Notice that many students are doing this at the same time. You may find that the static port number is often occupied by some other program(s). You had better bind to the next available port number rather than a static port.

4 System Termination

To gracefully terminate the system a client executes the function:

```
int rpcTerminate ( void );
```

The client stub is expected to pass this request to the binder. The binder in turn will inform the servers, which are all expected to gracefully terminate. The binder should terminate after all servers have terminated. Clients are expected to terminate on their own cognizance.

In a real system only privileged clients would be able to execute this function, the binder would be expected to authenticate the request, and the servers would authenticate the request from the binder. We will omit client authentication for the sake of simplicity, but we will have very crude binder authentication by the servers. Specifically, they should verify that the termination request comes from the binder's IP address/hostname.

5 Protocols

We now define a suggested message protocol. Strictly speaking, this is hidden behind the API just defined, and so you may choose any protocol scheme you wish. However, it is recommended that you at least understand it before trying your own technique.

There are several messages that must be sent and replied to for this system to function. In no particular order, they are the server/binder messages, the client/binder messages, and the client/server messages. Since messages are not quite the same as data structures, in that they do not have clear boundaries, it is strongly recommended that they take the following form:

```
Length, Type, Message
```

Where `Length` is an integer indicating the message length, `Type` is an integer indicating the type, and then the message follows. Thus, it is possible to read the first eight bytes to determine the length and type, and then know how much more needs to be read, and how to respond to it. In the following descriptions, we will only identify the type and message information. The type is in "all caps."

Note that function names are variable length strings but they should not exceed 64 characters in size.

5.1 Server/Binder Message

This message will need to identify the function and the argument types, and the server identifier (IP address or hostname) and port number so that the binder can register the procedure. The message will be:

```
REGISTER, server_identifier, port, name, argTypes
```

You would have to assume a fixed length for the IP address or hostname, port and name field in order to parse the message. The binder will respond with either REGISTER_SUCCESS or REGISTER_FAILURE, with an integer following both message types to indicate warnings or errors, if any.

5.2 Client/Binder Message

There is only one request message from clients to the binder, used to locate an appropriate server procedure. The message format is:

```
LOC_REQUEST, name, argTypes
```

where `name` and `argTypes` are the respective parameters from the rpcCall call. Again, note that you would need to assume a fixed length for the name field. On success, the binder will respond with a message of the form

```
LOC_SUCCESS, server_identifier, port
```

The server identifier and port indicate the port on which the server is listening for client requests. If the request failed, the binder will respond with a message of the form

```
LOC_FAILURE, reasonCode
```

where `reasonCode` is an integer indicating a specific failure condition.

5.3 Client/Server Message

There is only one request message from clients to servers used to request execution of a server procedure. It requires the argument types and argument values; the message format is

`EXECUTE, name, argTypes, args`

The response is

`EXECUTE_SUCCESS, name, argTypes, args`

upon successful execution and

`EXECUTE_FAILURE, reasonCode`

for failure, where `reasonCode` is an integer representing the reason for failure.

5.4 Terminate Messages

To terminate the servers and binder a client sends a terminate message of the form:

```
TERMINATE
```

to the binder. The binder sends the same message to all servers which, after verifying it is from the binder, terminate. It is not necessary, for the purposes of this assignment, to deal with this any more cleanly than as is described.

6 Bonus Functionality (worth 10% of the weight of Assignment 3)

6.1 Requirements

As a bonus feature, you will have to implement a cache system in client side library. The rpc client library would cache the server locations it received from the binder. The advantage of this scheme is that the client won't need to send a location request for every rpc request. The specific details are given in Section 6.1.1 and 6.1.2

Note: The bonus feature is intended as an additional piece of functionality and would be evaluated seperately. Please make sure that your implementation of this functionality in no way modifies the "default" behavior of your rpc system as described in Sections 1–5.

6.1.1 Binder

The binder would implement a cache location request, similar to the location request. But instead of just returning a single server's location for a particular function signature, it will return all the servers which have registered functions for the requested function signature. The implementation of this feature and the internal protocol you use is completely up to you, but make sure that this request is completely seperate from the location request.

6.1.2 Client

The client would need to implement an rpcCacheCall interface. The signature for rpcCacheCall is as follows (the same as rpcCall):

```
int rpcCacheCall(char * name, int * argTypes, void ** args);
```

It differs from rpcCall in the following ways:

1. `rpcCacheCall` caches the result of cache location request (i.e. mappings of function signature and server locations) in a local database similar to the database maintained by the binder.
2. For each request the client makes using rpcCacheCall, the local database would be scanned for a server matching the function signature. If a server is found, the client would directly call the server and receive the results. If the results are invalid or the server no longer exists, it would send the request to the next server in the local cache and so on. If the request fails on all servers in the local cache or there were none to begin with, it would transparently send a location cache request to the binder and get an updated list of servers for that particular function signature. It would cache the server locations and repeat the process by sending request to one server at a time, resulting in either success if a server replies with the result or failure in the case where all servers are exhausted without success.

The rpc library should only cache results and operate in this mode when the client uses rpcCacheCall. For the clients using rpcCall the behavior would be unchanged i.e. for every request, the rpcCall would first send the normal location request, get a server and then send the request to that particular server.

7 Requirements

7.1 Code with Makefile and README

You are required to implement this RPC system as described. In particular, you are required to implement the binder and the RPC library. You can only use C/C++ to implement it. You should submit all the code using the `submit` command on the student.cs environment.

To compile the client we will execute the following command:

```
g++ -L. client.o -lrpc -o client
```

And likewise for the server. The functions must be in a library called `librpc.a`. If your code needs other libraries, you should document it in the `README` file. Do NOT make any modifications to the rpc.h file. Create a separate header file if you need to make any other declarations. We shall only copy the rpc library and rpc.h on the client and server sides. No other header file will be copied there.

You are also required to write a makefile. By typing `make` it should generate the RPC library and the binder executable, named 'binder'.

Write a `README` file, describing how to compile and run your RPC system; also document any dependencies or other things. Remember to include the names AND userids of both group members in the `README`.

Please test your code on the machines in the `linux.student.cs` environment[1] before submission. If your makefile does not create the library or binder, or our clients/server do not compile with your library on this environment, we shall apply an automatic 10% penalty.

[1]Individual machines in this environment are `ubuntu1604-[002, 004, 006, 008].student.cs.uwaterloo.ca`

7.2 Documentation and System Manual

You will be required to submit a system manual with your RPC system. A .pdf should be submitted using the `submit` command with your code.

The system manual should include at least the following items. You can add other stuff if you wish but please be concise.

1. You should discuss your design choices so that we can understand how important functionalities were accomplished. For instance, you should discuss marshalling/unmarshalling of data, structure of your binder database, handling of function overloading, managing round-robin scheduling, and termination procedure. If there were other optimizations in your code/design that you think we should be aware of, feel free to list them in this section.

2. List all error codes that you have identified and very briefly describe what each error refers to.

3. Clearly identify any functionality of the assignment that has not been implemented.

4. If you have implemented something else or you want us to know about any advanced functionality, describe it here, along with a brief description of how that functionality can be tested. Do not provide any testing code here.

7.3 Evaluation

We will take the following steps to test your code with our own client and server:

1. `make` to compile your code to get librpc.a and binder executable in the linux.student.cs environment

2. `g++ -L. client.o -lrpc -o client` for compiling our client

3. `g++ -L. server_functions.o server_function_skels.o server.o -lrpc -o server` for compiling our server

4. `./binder`

5. Manually set the `BINDER_ADDRESS` and `BINDER_PORT` environment variables on the client and server machines. (Use `setenv` if using C shell)

6. `./server` and `./client` to run our server(s) and client(s). Note that the binder, client and server may be running on different machines.

Please ensure that the library and binder executable are compiled at the top level directory of your submission (e.g., they should not be compiled into a `/bin` directory).
Please also ensure that you submit ALL files necessary for compiling the library and binder.

While developing your RPC system, you should be aware that we shall be evaluating your system for the following:

1. Code compiling on linux.student.cs.

2. README file detailing:

    - any other dependencies for your library

- group member ids and names

3. Detailed documentation meeting the requirements in (Section 7.2).
4. We will be using a number of test cases to test the following functionality (though we may add or remove particular tests):
   - Basic functionality: `rpcInit, rpcCall, rpcRegister, rpcExecute, rpcTerminate`
   - Various data types including arrays
   - Binder and Server binding to dynamic ports
   - Blocking calls
   - Error checking including duplicate function registrations
   - Using appropriate error/reason codes for different failure/warning scenarios
   - Above functionality with multiple clients and/or servers
   - server/binder failure
   - server needs to handle invalid function calls

7.3.1 Specific Don'ts

1. **To avoid plagiarism, do not use any public or web-based source code repository hosting service other than** `https://git.uwaterloo.ca`.
2. The interface of rpcInit, rpcCall, rpcRegister, rpcExecute, rpcTerminate cannot change. We will be using these interfaces to run our server and clients. You may however develop your own protocol for internal communication e.g. the location request to the binder.
3. Do not modify rpc.h – instead create a different header file.
4. Do not bind to static ports or assume that the server/binder is running on the local machine.
5. Do not assume that the code will compile on linux.student.cs without actually compiling it on that environment.

7.4 Useful Resources

For those who are not familiar with the process of creating a static library, you can check the following link.

`http://tldp.org/HOWTO/Program-Library-HOWTO/introduction.html`

In coding this assignment, you may find useful to use a debugger, e.g., gdb is available on CSCF environment; the gdb user doc can be found at

`http://sources.redhat.com/gdb/onlinedocs/gdb.html`

POSIX threads (pthreads) is a standardized interface for threads on UNIX systems and a great tutorial with code examples can be found here

`https://computing.llnl.gov/tutorials/pthreads/`

A more general tutorial on threads including details of different threading interfaces is given in

`http://www.cs.cf.ac.uk/Dave/C/node29.html`

It is important that you start this assignment early!