Remote Procedure Call

CS3411 Fall 2024
Program Five

Due: Friday December 6, 2022, Midnight

In this project, we will develop a mini *Remote Procedure Call* (RPC) based system consisting of a *server*, and a *client*. Using the remote procedures supplied by the server our client program will be able to open files and perform computations on the server. Start early on this project and make sure that you follow a disciplined approach outlined in this assignment.

The Concept

Remote procedure technique provides procedures as a set of abstractions which hide the fact that the actual operation takes place at a remote computer and behave identically to familiar procedure call interface. To provide this functionality, they typically pack a procedure identifier followed by one or more arguments into a character buffer one after the other, and send the buffer using a socket/stream connection to the server. This packing operation is called the *Marshalling of the arguments*. The server, upon receiving the message, *unmarshalls* the arguments, i.e., creates identical copies of the original arguments locally. It then performs a local procedure call, packs the result from that call into a message and sends it back to the client. The client then unpacks the result and returns it to the caller, giving the illusion of a local procedure call.

The server

The server should open a socket and listen on an available port. The server program upon starting should print the port number it is using on the standard output. In order to implement the RPC, the server and the client should communicate through a TCP socket. It is allowed to fork a child for each open connection and delagate the handling to the child.

The server must support remote calling of the following functions:

open, close, read, write, seek. The behavior of these procedures should be identical to their kernel counterparts in terms of their arguments and return values. In order to distinguish them from kernel functions, we will add the prefix *rp_* to each of the names, e.g., rp_open.

The client

You are expected to develop a client program which will provide an environment into which we can *plug* an application and execute it using the remote versions of the supported calls. The client program therefore should expect a `<hostname> <portnumber>` pair as its arguments and attempt to connect the server. Once it connects, it should call the *user program*. This seamless usability will be provided by using a different function as the entry point, called *entry* instead of the usual *main*.

The client program's entry function should strip off the first two arguments, create a new argv array and call the main procedure. Finally, when *entry* exits, it should call the exit kernel function. When the user links her program(s) with your client, they should use the flags *-e entry* so that the linker will start the program with client's entry point, which in turn calls the main function of the user.

The client

Remote procedure calls can be implemented by providing a jacket function which has identical arguments to the (local) function it is implementing. For example, one could implement rp_open as follows (all error checking omitted, and it leaks):

```
int rp_open(const char *pathname, int flags, int mode)
{
 int    len;
 char * msg;
 int    in_msg;
```

```
 int    in_err;

 len = 1                 +       // this is the opcode
       sizeof(mode)      +       // int bytes for mode.
       sizeof(flags)     +       // int bytes for flags.
       strlen(pathname)  + 1;    // null terminated pathname

 msg = malloc(len);

 len=0;
 msg[len++] = 1;                 // this is the code for open.

 bcopy(&flags,&msg[1],sizeof(flags));    len+=sizeof(flags);         // move the value of flags.
 bcopy(&mode,&msg[1],sizeof(mode));      len+=sizeof(mode);          // move the value of mode.
 strcpy(&msg[len],pathname);             len+=strlen(pathname)+1;    // put the pathname.

 // This is where you can split (A).

 write(socketfd,msg,len);
 read(socketfd,&msg,8);

 bcopy(&in_msg,msg   ,sizeof(int));
 bcopy(&in_err,msg[4],sizeof(int));

 errno = in_err;
 return in_msg;
}
```

Note that there will be a symmetric component at the receiving end that will actually execute the open kernel call. Now the client program can call the remote procedure as if it is local:

```
 rslt = rp_open("myfile",O_CREAT | O_APPEND, 0600);
 if  (rslt < 0) perror("open failed");
```

Interoperability

Your server and your client should work interoperably with any others. Therefore, we need to use a uniform communication protocol. In order to keep things simple, we shall assume that both the remote host and the local host has identical architectures. In other words, binary data can be exchanged without a need to translate it to network byte orders.

Message formats

There are two types of messages, namely, request and response. All request messages have the form :

```
opcode fixed arg1, ... fixed argn, variable arg1, ... variable argn
```

Both fixed length arguments (fixed arg1 ... fixed argn) and variable length arguments should follow a left to right order given the function prototype. For example, open prototype has the first argument as a variable length argument(path), followed by flags and mode both fixed length arguments. Therefore, marshalling will first store the opcode, followed by the value of flags argument, then the mode argument and finally the path argument.

The opcode is a binary value representing the function to call:

```
#define open_call      1
#define close_call     2
#define read_call      3
#define write_call     4
#define seek_call      5
```

Response messages are of the form:

```
result errno data
```

where result is an integer (32 bits) returned by the system call, errno is an integer (32 bits) set at the remote host, and data is the data returned by the executed system call (only read in the above list).

The client program

The client program is a shell that we should provide the user of our RPC system to make her job of using the remote procedure calls easy. The idea is simple. The user simply develops a standalone program and tests it using a set of defines called rpctest.h:

```
#define r_open  open
#define r_close close
..
```

A simple program which opens a remote file, reads 20 characters and prints it can be written as below:

```
int main(int argc, char **argv)
{
int fd,r;
char buf[20];

fd = rp_open(argv[1], O_RDONLY, 0600);
r = rp_read(fd,buf,20);
printf("%s\n",buf);
return 0;
}
```

Now the user can compile the program by including rpctest.h. When the resulting binary is executed:

a.out myfile.txt

each call is executed using the kernel versions and we are sure the program works just fine. At this point, the user deletes the include reference rpctest.h and instead includes client.c. Compiled program this time is executed as:

a.out colossus.it.mtu.edu 2000 myfile.txt

Client's entry routine initiates a socket connection to the remote server and once the connection is established, calls the user's *main* passing the rest of the arguments, providing seamless operation remotedly.

Project Part 1

We can built these programs by either building all RPC code as a local code, test it and then move part of the code to the server side, or, develop our server, test it using simple interaction and implement the code piece by piece on the server (and the client) side.

In this project, we'll follow the former. For example, considering the rp_open case above, one can write the server side function which only takes a char * argument, extracts open arguments and calls open, forms a message and returns that message. If such a function is placed where the write to and read from the socket

descriptor take place in the above code, (marked as (A) in the example) we can test the functionality of the code locally without dealing with various networking issues. Once all functions have (local) implementations of the remote version, we can start playing with the server and move the processing functions there.

As a result, follow these steps :

1. Develop rpctest.h which is a simple file that defines which map remote versions of procedure names to local versions as illustrated above.
2. Copy the source code of the example, and test it with your rpctest.h.
3. Develop a file called client.c. The main program of client.c is called *entry*, and it should strip of the first two arguments, print them to stdout and call the function *main*, passing newly formed argv array that does not contain the first two arguments. Compiling client.c together with the example program should work when we execute the program by using two additional dummy arguments (e.g., a.out blah blah myfile.txt).
4. Add a prototype function rp_open into client.c as an empty function, delete the mapping from rpctest.h for rp_open such that it simply calls kernel call open by passing one to one its arguments. The program should still run correctly as above.
5. Add marshalling code for rp_open so that it packs its arguments to a message and calls a function called server which returns a char string and accepts a char string as an argument. Develop *server* function so that it unmarshalls the message passed as its argument into local variables, calls the kernel function open, packs the result back into a string and return that to the caller. rp_open unpacks the result and returns the values. Test the program. Make sure it works.
6. Repeat steps 4 to 5 for each of the to be implemented functions.

Once the above steps have been completed successfully, we need to implement a client-server pair and move part of the unmarshalling code as well as the functions which implement server functions to the server side. You can follow these steps to ensure testability during development:

1. Implement a client/server pair as discussed in class. You can use the code inet_wstream.c (client) and inet_rstream.c (server). Make sure you can connect to the server running on your localhost using these programs. These programs have all the functionality you will need for the first steps.
2. Put only one of the functions to be implemented on the client side, such as rp_open, move the unmarshalling code to the server side, insert a socket write on the client side to send the marhalled string to the server and print the arguments received on the server side. It is important that the client writes marshalled string in full and the server reads it in full to maintain the synchronous behavior.
3. Once you can perform one function call on the server side, introduce a switch statement on the server side. The server should read one byte from the socket and call the function corresponding to that code. Make sure the software still functions correctly.
4. Now move each of the functions from the client side to the server side, adding each opcode to the switch statement. The server side code will look like this:

```
...
code for accepting connections and connecting to the client.

while (1)
{
  l = read(sd, buf, 1)
  switch(buf[0])
  {
      case open_call : handle_open (sd);
```

```
                        break;

      case close_call : handle_close (sd);
                        break;
     ...
  }
}
```

5. Once all the functionality has been moved, your client.c should work with remote calls. Test your client with two *user* programs, one which remotedly opens an output file, locally opens an input file and copies the input file to the output file and close both files. This program when linked with the client framework should yield a binary *rclient1*. The second program should open a local file as output and a remote file as input. It should seek the remote file to position 10 and copy the rest to the local file. This program when linked with the client framework should yield a binary *rclient2*. Note that multiple clients can be in execution at any point in time.

6. When the above programs work correctly, look at inet_rstream2.c. Modify your server so that it can work with multiple clients by forking and delegating the functionality to a child process for each connection. Note that, the bulk of your grade will come from a correctly functioning client/server implementation described above. Ability to function with multiple clients should be the last thing you should attempt.

7. Submit your client.c, server.c, Makefile, rclient1.c and rclient2.c and any other files necessary for successful compilation of these programs as a tar file. See the details in Section *Submission Requirements*.

Ground Rules and Restrictions

1. You may discuss the program with others.

2. You may not bring any printed/written material into the discussion with you. (You may not show anyone your code; you may not view the code of anyone else. This includes others both enrolled in the course and others not enrolled in the course.)

3. You may generate written material during the discussion, but it must be destroyed immediately afterwards. Don't carry anything away from the discussion.

4. If you are in doubt about the acceptability of any collaboration activity, I expect you to check with me before engaging in the activity.

Sharing

It is permitted to test your client against another student's server and vice versa - provided that no source files are shared. For example, one of your friends can fire-up their server on their lab computer and tell you the hostname/port number information. You can then execute your rclient1 against their server. If there is a failure in your client you should debug it without your friend looking at your code. Similarly, if there is a failure at the server, it is your friend's call to do the debugging. You may also exchange binaries to further debug the problem, and to continue working on the problem alone, provided that there is absolutely no sharing of source code at any time.

Submission Requirements

Your submission must be written in C. Use Canvas to submit a tar file named `prog5.tgz` that contains:

1. A copy of the source files **with comments**.
2. A makefile with *all*, *rclient1*, *rclient2*, *rserver* and *clean*.
3. A README file which describes what works and what does not and a brief description of any of the tests you have performed.