Remote Procedure Calls

CS3411 Fall 2021
Program Four
Due: Sunday Nov 28, 2021, 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.

The server

The server should open a socket and listen on an any available port. You may not assume a fixed port is available The server program upon starting should print the port number it is using on the standard output. The only output from the server on stdout should be the port number printed as an integer. This port number is then manually passed as a command line argument to the client to establish the connection. 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, pipe, dup2.

In order to be able to use local versions of these calls together with the remote versions, each remote procedure will be prefixed with r_. For example, calling the kernel function open with its normal arguments as :

```
fd = open("myfile", O_RDONLY, 0600);
```

will call the kernel function as usual. If the remote version is to be called, it will be called as:

```
fd = r_open("myfile", O_RDONLY, 0600);
```

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 first two arguments and attempt to connect the server. Once it connects, it should call the *user program* which has a function called *entry* analogous to the main program in an ordinary C program. The entry routine should have the argv and argc arguments and return an integer value, just like the ordinary main. The client program should strip off the first two arguments, create a new argv array and call the entry procedure. Finally, when *entry* exits, the return value should be returned from the main procedure of the client.

When the user links her program(s) with your client, the combined program should compile into a workable binary. Test your client with two *user* programs, one which remotely opens an output file, locally opens an input file, copies the input file to the output file and closes both files. This program is called *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 is called *rclient2*. The input and output file names for *rclient1* and *rclient2* must be given as command line arguments. Assume the following order of arguments.

```
./rclient <hostname> <portnumber> <input_file> <output_file>
```

Note that multiple clients can be in execution at any point in time.

Implementing Calls

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 r_open as follows (all error checking omitted, and it leaks):

```
int r_open(const char *pathname, int flags, int mode)
{
 int    L;
 char * msg;
 char * p;
 int    in_msg;
 int    in_err;
 int    u_l;

 p=pathname;
 while(*p) p++;

 u_l = p-pathname;
 L   = 1     +                // this is the opcode
       2     + u_l  +         // 2-byte length field followed by the pathname.
       sizeof(flags) +        // int bytes for flags.
       sizeof(mode);          // int bytes for mode.

 msg = malloc(L);
 L=0;
 msg[L++] = 1;                                      // this is the code for open.
 msg[L++] = (u_l >> 8) & 0xff;                      // this is the length.
 msg[L++] = (u_l) & 0xff;

 for (i=0; i < u_l; i++)
 msg[L++]=pathname[i];                              // put the pathname.

 msg[L++] = (flags >> 24) & 0xff;                   // put the flags.
 msg[L++] = (flags >> 16) & 0xff;
 msg[L++] = (flags >>  8) & 0xff;
 msg[L++] = (flags      ) & 0xff;

 msg[L++] = (mode >> 24) & 0xff;                    // put the mode.
 msg[L++] = (mode >> 16) & 0xff;
 msg[L++] = (mode >>  8) & 0xff;
 msg[L++] = (mode      ) & 0xff;

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

 write(socketfd,msg,L);

 read(socketfd,&msg,8);

 in_msg = (msg[0] << 24) | (msg[1] << 16) | (msg[2] << 8) | msg[3];
 in_err = (msg[4] << 24) | (msg[5] << 16) | (msg[6] << 8) | msg[7];

 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 = r_open("myfile",O_CREAT | O_APPEND, 0600);
if  (rslt < 0) perror("r_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.

Message formats

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

```
opcode arg1, arg2, ... argn
```

All integer arguments are sent in big-endian form as four bytes. All variable length data such as character strings are preceeded by a two byte (short int) length field which is also sent in the big-endian byte order. The opcode is a one byte 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
#define pipe_call   6
#define dup2_call   7
```

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 returns data and the result encodes its length).

Approaching the problem

You 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 your server, test it using simple interaction and implement the code piece by piece on the server (and the client) side. My recommendation is to follow the former. For example, considering the r_open case above, one can write the server side function which only takes a char * argument, extracts the arguments to open, 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, you can start playing with your server and move the processing functions there.

Ground Rules and Restrictions

Grading

Your program should implement at least part of this functionality correctly to get any points. The criteria is given below:

1. Server program successfully creates a socket and listens for connections.
2. Client program successfully connects to a host/port pair.
3. Each remote function tested correctly receives input arguments.
4. Each remote function tested correctly returns the results.
5. File copy to and from the remote hosts work correctly.
6. Program(s) are interoperable.

Pragmatics

Source Template
You are provided a source template that includes:

- Makefile with the following commands:
  - all - compile project into *r_server, r_client.o, rclient1, rclient2.*
  - r_server - compile r_server only
  - r_client - compile r_client library only
  - rclient1 - compile rclient1 and link against r_client
  - rclient2 - compile rclient2 and link against r_client
  - clean - removes executable and object files.
  - submission - generates prog4.tgz with all source files needed for submission. Upload this file to Canvas.
- r_server.c - C source for implementation of the server
- r_client.c - C source for implementation of the RPC calls and main()
- rclient1.c - C source where you must implement user program 1
- rclient2.c - C source where you must implement user program 2

Submission Requirements

Your submission must be written in C. Use Canvas to submit a tar file named prog4.tgz generated by running *make submission.*

Use Canvas to submit a tar file named `prog4.tar` that contains:
When I execute the commands: `tar xvf prog4.tar; make` against your submission, executables named `server`, `rclient1`, `rclient2` should be created in the current directory. Hardcopy of your code is not needed.