Fall 2015 **EECS 338 Assignment 6** **Due:** November 19, 2015

G. Ozsoyoglu *RPC programming*

Cookie Jar Server Problem; 100 points

In this assignment, you will implement a simple, connection-oriented, RPC-based mailbox server and its clients with Linux remote procedure calls (RPCs). For this assignment, in addition to *eecslinab3*, two more virtual servers are created, namely, *eecslinab2* and *eecslinab7*, created with exactly the same configuration. To make sure that you can access all three servers without any problems, make sure to log into these two servers.

The (revised) Finite-Source Cookie Jar Problem. A cookie jar is being shared by two sisters, Tina and Judy, using the following rule: When Judy requests from her mother a cookie from the jar, Judy gets a cookie only after Tina (being the older sister) gets a cookie in at least two separate occasions, whereas Tina requests and gets a cookie from the mother whenever she wants to (i.e., if Judy wants a cookie, the mother has to make sure that Tina has had two cookies since the last time Judy got one). The cookie jar has finite number of cookies (in this implementation, only 20 cookies). Tina and Judy can get only one cookie each time they request cookies. Whenever the jar is empty, and Judy (or Tina) requests a cookie, the mother tells that, sadly, the jar is empty and "*no more cookies for you*".

Treating Tina, Judy and the mother as three independently executing processes, give an RPC-based solution to this problem.

- Your solution consists of three programs, one for Tina (client), one for Judy (another client), and one for the mother (server).
- The cookie jar has at the beginning exactly 20 cookies.
- Judy and Tina request, at random times, from the mother process (server) a cookie (to be taken out, by the mother, from the cookie jar), and eat it.
- Make sure that Judy and Tina processes keep requesting cookies, until the cookie jar becomes empty, at which time the mother informs both Judy and Tina that there are no more cookies, and then everybody terminates.
- For each action, print a message to the screen specifying the action, the requesting child name, and containing the names of the involved processes.
- There will be one Tina process (client), one Judy process (client), and one "mother" process (server), Judy and Tina each running on a distinct machine, and the mother running on the third machine.
- Judy and Tina processes both call only one remote procedure, namely, *GetMeMyCookie (a, b)*: the parameter *a* returns
    - −2 if the cookie jar is empty;
    - −1 if it is Judy and Judy cannot get a cookie at this point of time;
    - +1 if the cookie is successfully returned.
- The parameter *b* indicates whether it is Judy or Tina who is requesting the cookie.

Make sure to

1. Copy **all** your code to each server,
2. Compile all of it on each server,
3. Start the mother (i.e., server) process (before starting the two clients, i.e., Judy and Tina), and
4. Start each client (i.e., Judy and Tina) process on the other two servers.

Test your program, script, and (a) turn in the output as well as your source code, (b) upload your assignment (source code, binary, and the printout) to the blackboard. The clients should print their actions with the machine name and the time attached.

There are three programs you have to write:

1. Definition of the RPC protocol in the file, say, *Cookie.x*
2. The service program in the file, say, *Mother.c*. This program contains the actual procedure to be invoked by the server dispatch routine in the compiler-generated stub. Please note that you can define and use several remote procedures to be called with RPCs using a single pair of client/server stubs.
3. Two client program files, say, *Judy.c* and *Tina.c*. Each client code should create a server handle, the attachment to the appropriate remote services of servers.

Application development with RPCGEN: The protocol definition file *Cookie.x* is first processed by RPCGEN (by issuing the command "RPCGEN Cookie.x"), the RPC stub generation compiler. RPCGEN generates:

(a) XDR wrapper routines in *Cookie_xdr.c* (which are bidirectional filters used by both the client and the server),
(b) the associated common "#include"s in *Cookie.h*,
(c) client stub in *Cookie_clnt.c,*
(d) server stub in *Cookie_svc.c*. Please note that the server stub has a main. Thus, the remote procedure that you define in *Mother.c* should not have a main since it will be linked with the server stub.

The Protocol Definition Language: This is the language used for automated stub generation. The code will be stored in the *Cookie.x* protocol definition file. Here is an example of a program definition:

```
program APPLICATION_PROGRAM                    /* Can specify multiple servers */
{version APPLICATION_VERSION
  {struct output-parameter1 REMOTE-PROCEDURE1 (input-parameter1) = 1;
   } = 1;                          /* RPC program version number (used for bookkeeping) */
} =22855                           /* program number (used for bookkeeping) */
```

Note that the client RPC call *must* have as the last parameter the server handle (created by the *clnt_create* system call) as a call-by-reference parameter. This parameter is only used by client and server stubs, not by the application. Also, all remote procedures should have the suffix _1 (as the version number) in their names.

At the Client: A client is required to maintain a unique structure (pointed to by a server handle) for each client/server connection. And, the "#include"s that you need to use (a) in the client, say, *Judy.c* is "Cookie.h", and (b) in the server are <rpc/rpc.h> and "Cookie.h".

The RPC system calls that you need to use are *clnt_create* (you should use the UDP protocol (as opposed to TCP) in the specification) and *clnt_pcreateerror* and *clnt_destroy*.

At the Server: The server procedures that you write will be in *Mother.c* . The "#include" that you need to use in *Mother.c* is "Cookie.h".

To compile and run RPCGEN: You can compile the protocol definition with the following command:

% rpcgen Cookie.x

Then, RPCGEN produces a client stub (*Cookie_clnt.c*), a server stub (*Cookie_svc.c),* XDR filters (*Cookie_xdr.c*), and the header file (*Cookie.h*).

You can compile the client code for, say, the *ClientA* and link it with the client stub and the XDR filter generated by RPCGEN using the command

%gcc -o ClientA Client.c Cookie_clnt.c Cookie_xdr.c –lnsl

You can compile the server code and link it with the server stub and the XDR filter generated by RPCGEN using the command

%gcc -o Server Cookie_svc.c Cookie_xdr.c Server.c -lnsl

To start remote processes: You can login to three different machines, and start first the server (i.e., Mother), and then the two clients (i.e., Judy and Tina). Or, you can use the remote shell command to start the server and the clients. Below is a shell script to launch the server Mother on the remote host *eecslinab3*.

ssh -f cwru123@eecslinab3 "./path/to/files/cookieserver"

where *cwru123* is your cwru e-mail id. There is no need for '&', since the '−*f*' flag tells *ssh* to go into the background after getting the password. Combining these two ideas, you can start all programs on different hosts:

ssh -f cwru123@eecslinab3 "./cookieserver/mother"; ssh -f cwru123@eecslinab2 ".cookieserver/Tina";
ssh -f cwru123@eecslinab7 ".cookieserver/Judy"

So this runs "*mother*" *on eecslinab3, "Tina"* on *eecslinab2*, and *"Judy"* on *eecslinab7*.

Note: The RPC coverage in the recitation and the class, together with the man pages of the Linux OS, should be sufficient for this assignment. For a set of Linux RPC examples (with RCGEN and makefiles), see http://www.linuxjournal.com/article/2204. Also, the book "Power Programming with RPC" by J. Bloomer, O'Reilly & Associates, 1991 is an excellent source on RPCs, and has extensive RPC examples.