UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI
UNDERGRADUATE UNIVERSITY

Remote Shell using RPC (Multiple clients)

Final Report

Supervised By
Dr. Tran Giang Son

Submitted By

Nguyen Trac Thanh	USTHBI8-170
Le Quang Vinh	USTHBI8-188
Dao Anh Hong	USTHBI8-068
Cao Phuong Linh	USTHBI7-091
Dang Minh Duc	USTHBI8-047

Hanoi, April 2020

TABLE OF CONTENTS

I. INTRODUCTION

In this project, we perform Remote Shell using RPCs. We use RPCGEN because the code is simple, written in C which we are already familiar with, and tutorials are available online.

Remote shell (rsh) is a command line computer program that can execute shell commands as another user, and on another computer across a computer network. The remote system to which rsh connects runs the rsh daemon (rshd). The daemon typically uses the well-known Transmission Control Protocol (TCP) port number 514.

II. OBJECTIVE

The goal is to have one server that can serve multiple clients at the same time. Each client sends shell commands, the server executes it (on the server machine), and returns the output to the client.

III. METHOD

1. What is RPC?

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

2. How does RPC work?

(figure 1)

Step 1: Client machine making a RPC call to a server machine that registered to serve this specific call.

Step 2: Procedure parameters transferred across the network to the relevant server machine.

Step 3: Server machine execute the procedure.

Step 4: When the procedure finishes and results are produced, they are transferred back to the calling environment.

Step 5: The execution on the client machine resumes as if returning from a regular (local) procedure call.

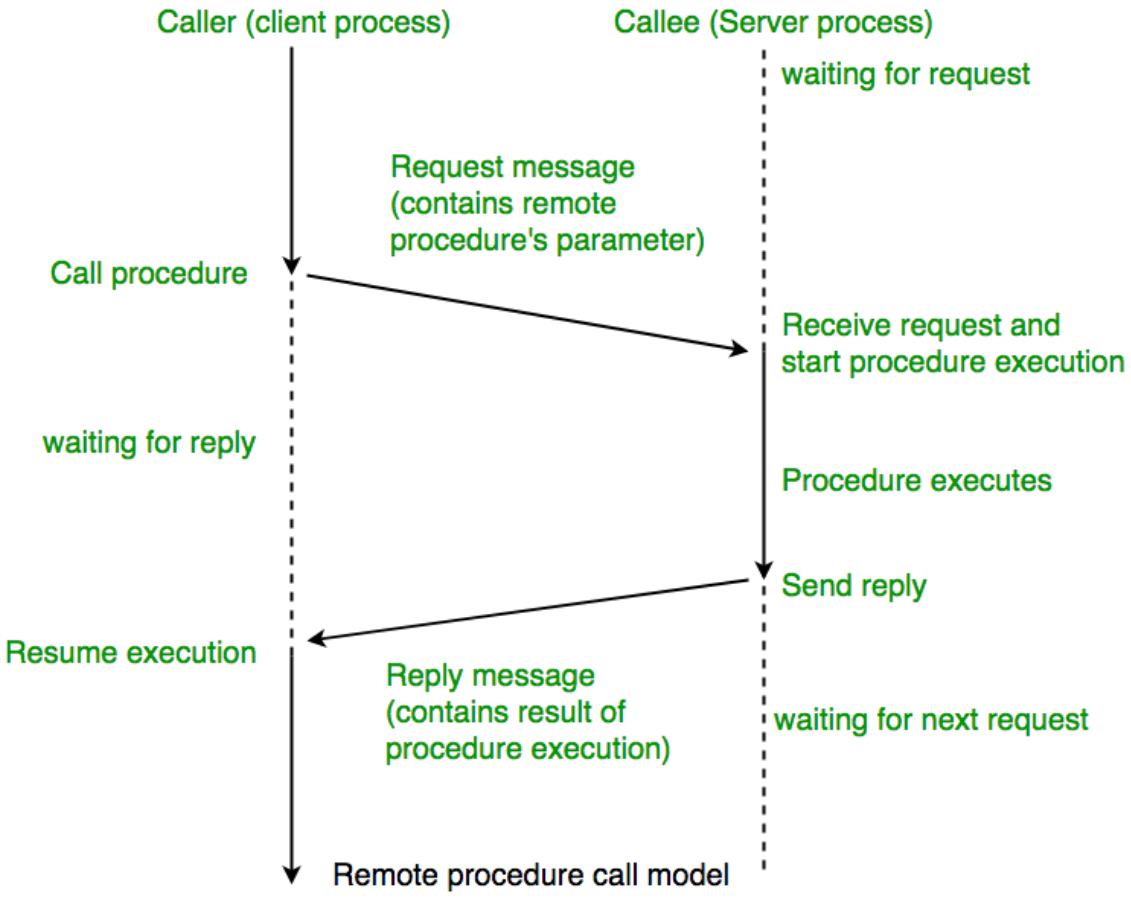

Figure 1.

3. Client/Server communication interface:

- We specify the RPC methods using rpcgen format in file “exec.x”.

- We only have 1 method that takes a string<> (<> means variable length) and returns a string. Note that in rpcgen, most things are pointer, so a string (char*) will be represented by char** in rpcgen-generated functions.

```
Class exec.x:
program EXECSH {
    version EXECSH_V1 {
            string EXEC_STRING(string<>) = 1;
    } = 1;
} = 0x12345678;
```

4. Server code generation:

- We use "rpcgen exec.x" to generate file exec.h, exec_clnt.c, exec_svc.c

- exec.h is the header file that declares all the functions we will use. You can see the function prototypes of both the client and server here.

- exec_clnt.c contains only the function that the client will call to execute RPC. We need to finish this file by implementing the main() function and add things such as console input/output.

- exec_svc.c contains the main() already done for us, and execsh_1(), which is a complicated function that handles network/RPC calls. We need to finish this file by implementing exec_string_1_svc(), which perform Shell command execution, and returns the result.

5. Client code:

- First, we create a CLIENT* object to connect to the server.

- After that, when we want to RPC, we only need to use exec_string_1(), which is generated by rpcgen.

- Client summary:
  User input string *s*
  Call RPC with *s* as input

Receive result and print (and handle error).

6. Server code:

- rpcgen functions always return pointer. So a string (char*) will return a char**.

- We have global variable **res to return the result.

- In exec_svc.c, we need to implement exec_string_1_svc(). Basically, it takes an input string, Shell execute that string using popen() which returns a file descriptor (FILE*) that allows us to read the output of the shell command.

- More detail in the code.

IV. RESULT

1. Example image:

- Build server

```
ds_code$ cd RPC_exec
ds_code/RPC_exec$ cd server
ds_code/RPC_exec/server$ chmod +x makeServer
ds_code/RPC_exec/server$ ./makeServer
ds_code/RPC_exec/server$ ./server
```

- Client build and connect to server(localhost)

```
RPC_exec/client$ chmod +x makeClient
RPC_exec/client$ ./makeClient
RPC_exec/client$ ./client 192.168.1.10
```

- Client and server are connecting

```
Connection ready. Please type shell commands
```

- Client push command to server

```
>ls -a
```

- Server receive client's command

```
Client ask: ls -a
```

- Client receives server's output

```
exec.h
exec_svc.c
makeServer
server
```

2. Evaluation:

- The server gives correct output, that can have unlimited size
- The server can serve multiple different clients. Test by launching 2 ./client in 2 different folder.
- No user error handling: if the user's input is incorrect, server will output an error but the user receives an empty string
- Can't change server directory.

V. CONCLUSION

Our Remote Shell program works and can support many clients at once. However, error handling needs to be improved.

REFERENCES:

- https://www.geeksforgeeks.org/remote-procedure-call-rpc-in-operating-system/
- http://users.cs.cf.ac.uk/Dave.Marshall/C/node33.html
- https://www.cprogramming.com/tutorial/rpc/remote_procedure_call_start.html