# Department of Computer Science

**EE353: Computer Networks**

**Name:** M. Hasnain Naeem

**Reg #:** 212728

**Class:** BSCS-7B

# Lab 10: **Socket Programming (Remote Procedure Calls)**

# **Date: 28th Oct, 2019**

**Lab Engineer: Engr. Kaleem Ullah**

# Instructor: Dr. Muhammad Zeeshan

# Lab 10: Socket Programming for Linux

**Introduction**

**Remote Procedure Calls**

Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC makes coding easy in distributed systems’ applications since the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. That is, the programs are coded as if the procedure call were a normal (local) procedure call, without explicitly coding the details for the remote interaction.

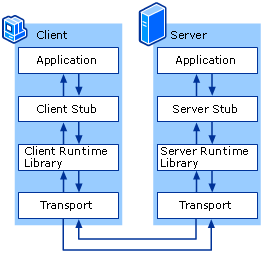
Message passing in RPC

The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.

* The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
* The client’s local operating system sends the message from the client machine to the server machine.
* The local operating system on the server machine passes the incoming packets to the server stub.
* The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
* Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Steps involved in developing a RPC application developing:

* Specify the protocol for client server communication
* Develop the client program
* Develop the server program



##### **Step1 : Specify the protocol for client server communication**

##### An interface description language (IDL) to let various platforms call the RPC. The IDL files can then be used to generate code to interface between the client and servers. So, in IDL file, we define the program structure like below. Save this IDL file with .x extension.

calculate.x

struct inputs{

float num1;

float num2;

char operator;

};

program CALCULATE\_PROG{

version CALCULATE\_VER{

float ADD(inputs)=1;

float SUB(inputs)=2;

float MUL(inputs)=3;

float DIV(inputs)=4;

}=1;

}=0x2fffffff;

inputs — Name of the data structure. Through this data structure the parameters are send to the server for computations.

CALCULATE\_PROG — Name of the program

CALCULATE\_VER — Name of the program version

ADD(inputs) — This is a remote method which is calling locally and the parameters are passed to that remote method through the inputs structure which contains 2 operands(2 numbers) and an operator.

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number). Therefore, you need to give any numbers you like to the program, to the version and to the procedures as in the above sample IDL file.

Then compile your IDL file using **rpcgen** protocol compiler. The protocol compiler reads the definition of the IDL file and automatically generates client and server stubs. First you need to check whether the **rpcbind** has been installed in your machine. Type this command to check it and if the execution of the command gives you a long list , then your machine has already installed **rpcbind** package and you can work on with the **rpc** commands.

Try this command if it is already installed

$ rpcinfo

Otherwise, you have to install the **rpcbind** package by following below command and check whether it has been actually installed by typing the $ rpcinfo command again.

$ sudo apt-get install rpcbind

Now your PC is ready to execute rpc commands. The next step is to compile your IDL file using rpcgen command.

**$ rpcgen -a -C calculate.x**

This command generated 7 additional files. Keep in mind that, you can edit only the ‘calculate\_client.c’ and ‘calculate\_server.c’ files among from the created files according to your need. Don’t do changes to other files.

* calculate\_client.c —-> client program (editable file)
* calculate\_server.c —> server program (editable file)
* calculate\_cln.c —> client stub
* calculate\_svc.c —> server stub
* calculate\_xdr.c —> XDR(External Data Representation) filters
* calculate.h —> header file needed for any XDR filters
* Makefile.calculate —> compile all the source files by using this file

##### **Step 2 – Develop the client program**

edit the code in the calculate\_client.c file like the way it accepts the client’s inputs from the keyboard and outputs the remote procedures (add,sub,mul,div) invoking results to the client .

calculate\_client.c

#include "calculate.h"

float calculate\_prog\_1(char \*host,float n1,float n2,char opr,CLIENT \*clnt)

{

float \*result\_1;

inputs add\_1\_arg;

float \*result\_2;

inputs sub\_1\_arg;

float \*result\_3;

inputs mul\_1\_arg;

float \*result\_4;

inputs div\_1\_arg;

if(opr=='+'){

add\_1\_arg.num1=n1;

add\_1\_arg.num2=n2;

add\_1\_arg.operator=opr;

result\_1 = add\_1(&add\_1\_arg, clnt);

if (result\_1 == (float \*) NULL) {

clnt\_perror (clnt, "call failed");

}

return \*result\_1;

}

else if(opr=='-'){

sub\_1\_arg.num1=n1;

sub\_1\_arg.num2=n2;

sub\_1\_arg.operator=opr;

result\_2 = sub\_1(&sub\_1\_arg, clnt);

if (result\_2 == (float \*) NULL) {

clnt\_perror (clnt, "call failed");

}

return \*result\_2;

}

else if(opr=='\*'){

mul\_1\_arg.num1=n1;

mul\_1\_arg.num2=n2;

mul\_1\_arg.operator=opr;

result\_3 = mul\_1(&mul\_1\_arg, clnt);

if (result\_3 == (float \*) NULL) {

clnt\_perror (clnt, "call failed");

}

return \*result\_3;

}

else if(opr=='/'){

div\_1\_arg.num1=n1;

div\_1\_arg.num2=n2;

div\_1\_arg.operator=opr;

if(n2 == 0){

printf("Division by zero is not valid.\n");

exit(0);

}else{

result\_4 = div\_1(&div\_1\_arg, clnt);

if (result\_4 == (float \*) NULL) {

clnt\_perror (clnt, "call failed");

}

return \*result\_4;

}

}

}

int main (int argc, char \*argv[])

{

char \*host;

float a,b;

char op;

CLIENT \*clnt;

if (argc < 2) {

printf ("usage: %s server\_host\n", argv[0]);

exit (1);

}

printf("Welcome to Quick Cal!!!\n");

printf("+ for Addition\n- for Substraction\n\* for Multiplication\n/ for Division\n");

printf("Enter number 1 :\n");

scanf("%f",&a);

printf("Enter number 2 :\n");

scanf("%f",&b);

printf("Enter the Operator :\n");

scanf("%s",&op);

host = argv[1];

clnt = clnt\_create (host, CALCULATE\_PROG, CALCULATE\_VER, "udp");

if (clnt == NULL) {

clnt\_pcreateerror (host);

exit (1);

}

printf("The Answer = %f\n",calculate\_prog\_1 (host,a,b,op,clnt));

clnt\_destroy (clnt);

exit (0);

}

##### **Step 3 – Develop the server program**

Similarly you can edit the calculate\_server.c file while adding your own codes for the previously defined remote methods(add,sub,mul,div).

calculate\_server.c

#include "calculate.h"

float \* add\_1\_svc(inputs \*argp, struct svc\_req \*rqstp)

{

static float result;

result = argp->num1+argp->num2;

printf("Got Request : Adding %f and %f\n",argp->num1,argp->num2);

printf("Sent Response : %f\n",result);

return &result;

}

float \* sub\_1\_svc(inputs \*argp, struct svc\_req \*rqstp)

{

static float result;

result = argp->num1-argp->num2;

printf("Got Request : substituting %f from %f\n",argp->num2,argp->num1);

printf("Sent Response : %f\n",result);

return &result;

}

float \* mul\_1\_svc(inputs \*argp, struct svc\_req \*rqstp)

{

static float result;

result = argp->num1\*argp->num2;

printf("Got Request : Multiplying %f by %f\n",argp->num1,argp->num2);

printf("Sent Response : %f\n",result);

return &result;

}

float \* div\_1\_svc(inputs \*argp, struct svc\_req \*rqstp)

{

static float result;

result = argp->num1/argp->num2;

printf("Got Request : Dividing %f by %f\n",argp->num1,argp->num2);

printf("Sent Response : %f\n",result);

return &result;

}

After editing the client and the server code, you need to compile the files. As usual, it’s mandatory to compile the files if u made any changes. Otherwise those changes won’t apply at the execution time. Use this command for compilation of files.

$ make -f Makefile.calculate

This command will generate additional 2 executable files of the client and the server. (calculate\_client and calculate\_server)

To up the server,run the executable file of the server(calculate\_server) which was created at the compile time. To do that, type the below command in a terminal.

$ sudo ./calculate\_server

Take a new terminal. Then run the executable file of the client(calculate\_client) which was created at the compile time.

$ sudo ./calculate\_client localhost

Now the client program is ready to accept keyboard inputs from the client. So, you can enter 2 numbers and an operator which gives the output to the client by invoking the remote procedures at the server side which the client has requested.

**Example of Passing Command line Argument in C**

**command line arguments** are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly

#include <stdio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with single argument, it produces the following result.

**./a.out testing**

**The argument supplied is testing**

When the above code is compiled and executed with a two arguments, it produces the following result.

**$./a.out testing1 testing2**

**Too many arguments supplied.**

When the above code is compiled and executed without passing any argument, it produces the following result.

**$./a.out**

**One argument expected**

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and \*argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes

#include <stdio.h>

int main( int argc, char \*argv[] ) {

printf("Program name %s\n", argv[0]);

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

**$./a.out "testing1 testing2"**

**Program name ./a.out**

**The argument supplied is testing1 testing2**

**Task:** Design and code a sender/receiver program, client side should be able to get command from user and send that command to server over a TCP connection. Server must receive, execute that command and reply results to sender and results must also be displayed on client side screen.It is required to code in GNU C on Linux operating system.

**Specifications:**

Client will get command input from user and send it to server

Server will execute the command remotely on server machine and will reply back the results.

Client will display the execution results to the user on client's screen

Server should be able to receive and run following commands.

**Objectives:**

-Remote execution of commands sent by client on the server.

- Communication through the TCP protocol between server and client.

- Returning results from the client to server.

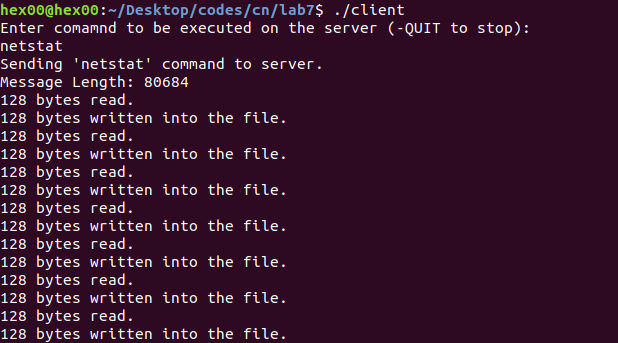
**Client.c**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  #include <netdb.h>  #include <sys/types.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <sys/socket.h>  #define PORT 5000  #define BUFFSIZE 128  #define RESULTSLEN 100000  int main(int argc, char \*argv[])  {      struct sockaddr\_in server\_details;      struct hostent \*he;      int socket\_fd;      int bytes\_read = 0;      long long total\_bytes\_read = 0;      char results[RESULTSLEN] = {0};      long long message\_length = 0;      char command\_buffer[BUFFSIZE] = {0};      char results\_buffer[BUFFSIZE] = {0};      int command\_len = 0;      char \*server\_address\_str = NULL; // sender address      // used to store and read the results      FILE \*results\_file;      char readC;      if (argc < 2)      {          // assume the server address          server\_address\_str = "127.0.01";      }      else if (argc >= 2)      {          server\_address\_str = argv[1];      }      if ((he = gethostbyname(server\_address\_str))==NULL)      {          fprintf(stderr, "ERROR: Could not get the hostname.\n");          exit(1);      }      if ((socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0))== -1)      {          fprintf(stderr, "Error: Coould not create the socket.\n");          exit(1);      }      // configuring the server details      memset(&server\_details, 0, sizeof(server\_details));      server\_details.sin\_family = AF\_INET;      server\_details.sin\_port = htons(PORT);      server\_details.sin\_addr = \*((struct in\_addr \*)he->h\_addr);      if (connect(socket\_fd, (struct sockaddr \*)&server\_details, sizeof(struct sockaddr))<0)      {          fprintf(stderr, "ERROR: Connection Failure.\n");          exit(1);      }      // To send multiple commands      int i = 0; // to iterate over the input      while(1) {          // comamand input          printf("Enter comamnd to be executed on the server (-QUIT to stop):\n");          fgets(command\_buffer, BUFFSIZE - 1, stdin);          command\_len = strlen(command\_buffer);          // removing the newline character from input          for(i = 0; i < command\_len; i++)          {              if(command\_buffer[i] == '\n')              {                  command\_buffer[i] = '\0';                  break;              }          }          // function compares both strings to given length (returns 0 if equal)          if (!strncmp(command\_buffer, "-QUIT", i+1))             break; // exit from program          printf("Sending '%s' command to server.\n", command\_buffer);          // sending commmand          if ((send(socket\_fd, command\_buffer, strlen(command\_buffer), 0))== -1)          {              // deal with erros              fprintf(stderr, "ERROR: Failed to send the command.\n");              close(socket\_fd);              exit(1);          }          else          {              total\_bytes\_read = 0;              bytes\_read = 0;              message\_length = 0;              recv(socket\_fd, &message\_length, 64, 0);              message\_length = ntohl(message\_length);              printf("Message Length: %lld\n", message\_length);              // keep on recieving the results until the recieved bytes < BUFFSIZE              results\_file = fopen("recieved\_results.txt", "w");              if (message\_length != 0)              {                  while(1)                  {                      bytes\_read = recv(socket\_fd, results\_buffer, sizeof(results\_buffer),0);                      printf("%d bytes read.\n", bytes\_read);                      if ( bytes\_read <= 0 )                      {                          printf("ERROR: Could not recieve results from the server. It can be due to closed connection or some other reason.\n");                          //Break from the While                          break;                      }                      // break when the exiting message is recieved                      if (total\_bytes\_read + bytes\_read >= message\_length) {                          fwrite(results\_buffer, 1, message\_length - total\_bytes\_read, results\_file);                          printf("Total bytes read = %lld\n", total\_bytes\_read+bytes\_read);                          break;                      }                      total\_bytes\_read += bytes\_read;                      // save results                      fwrite(results\_buffer, 1, strlen(results\_buffer), results\_file);                      printf("%d bytes written into the file.\n", bytes\_read);                  }                  // empty the buffer                  memset(results\_buffer, 0, BUFFSIZE \* (sizeof results\_buffer[0]) ); // to prevent saving previously stored data if read data < BUFFSIZE                  fclose(results\_file);                    printf("\nResults recieved from the server:\n");                  printf("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");                  // read results to console                  results\_file = fopen("recieved\_results.txt", "r");                  while((readC=fgetc(results\_file))!=EOF){ // printing character by character                      printf("%c", readC);                // because file can be large                  }                  fclose(results\_file);              }              else {                  printf("Command executed.");              }          }          printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");      } // while loop ends, user typed "-QUIT"      close(socket\_fd);  }//End of main |

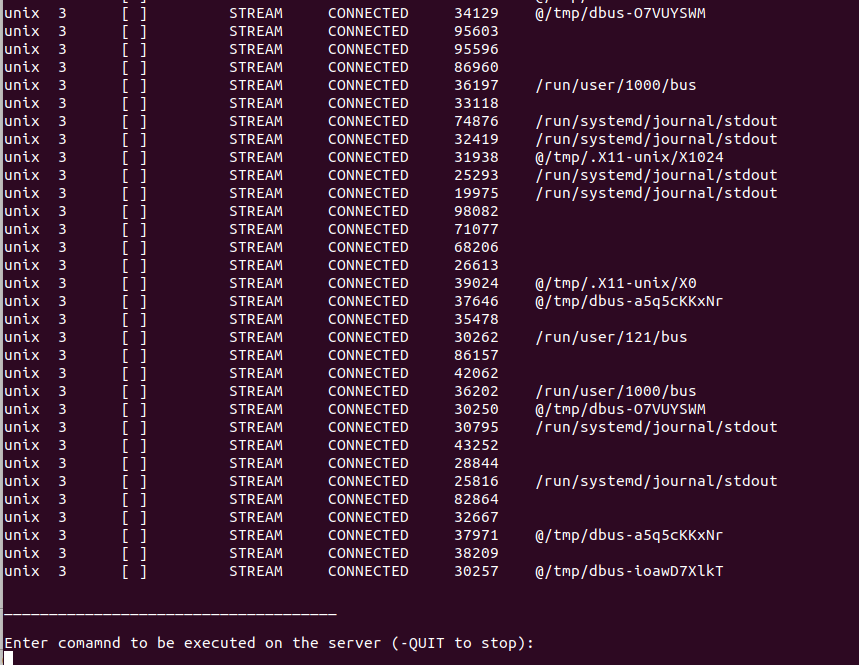
**Server.c**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  #include <sys/types.h>  #include <netdb.h>  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #define PORT 5000  #define BACKLOG 10  #define RESULTSLEN 100000  #define BUFFSIZE 128  #define PATHSIZE 100  int main()  {      // networking vars      struct sockaddr\_in server\_info;      struct sockaddr\_in client\_info;      int status, socket\_fd, client\_fd, num;      socklen\_t size;      char results[RESULTSLEN] = {0};      char command\_buffer[BUFFSIZE] = {0};      char results\_buffer[BUFFSIZE] = {0};      int reuse\_flag = 1; // true      long long message\_length = 0;        // used to run commands and store results      FILE \*fp;      FILE \*results\_file;                      // file pointer to temp results file      if ((socket\_fd = socket(AF\_INET, SOCK\_STREAM, 0))== -1) {          fprintf(stderr, "ERROR: Socket creation failure.\n");          exit(1);      }      if (setsockopt(socket\_fd, SOL\_SOCKET, SO\_REUSEADDR, &reuse\_flag, sizeof(int)) == -1) {          perror("setsockopt");          exit(1);      }      memset(&server\_info, 0, sizeof(server\_info));      memset(&client\_info, 0, sizeof(client\_info));        server\_info.sin\_family = AF\_INET;      server\_info.sin\_port = htons(PORT);      server\_info.sin\_addr.s\_addr = INADDR\_ANY;      if ((bind(socket\_fd, (struct sockaddr \*)&server\_info, sizeof(struct sockaddr )))== -1)    { //sizeof(struct sockaddr)          fprintf(stderr, "ERROR: Binding Failure.\n");          exit(1);      }      if ((listen(socket\_fd, BACKLOG))== -1){          fprintf(stderr, "ERROR: Listening Failure.\n");          exit(1);      }      while(1) {          size = sizeof(struct sockaddr\_in);          if ((client\_fd = accept(socket\_fd, (struct sockaddr \*)&client\_info, &size))==-1 ) {              perror("accept");              exit(1);          }          printf("Connection established with a client: %s.\n", inet\_ntoa(client\_info.sin\_addr));          while(1) { // Keep on dealing with the established connection              message\_length = 0;              if ((num = recv(client\_fd, command\_buffer, BUFFSIZE, 0))== -1) {                      perror("recv");                      exit(1);              }              else if (num == 0) {                  printf("Connection closed\n");                  // Proceed to deal with other client                  break;              }              command\_buffer[num] = '\0';              printf("Command Received: %s\n", command\_buffer);              /\* Begin Execution of Command and Store Results \*/              // command contains the command string (a character array)              // execute the command              fp = popen(command\_buffer, "r");              // file to write results              results\_file = fopen("results.txt", "w");              printf("Results are being sent to the client.\n");              // store the results in buffer and then into the file              while (fgets(results\_buffer, BUFFSIZE, fp) != NULL) {                  fwrite(results\_buffer, 1, strlen(results\_buffer), results\_file);              } // data transferred              fseek(results\_file, 0L, SEEK\_END);              message\_length = ftell(results\_file);              fclose(results\_file);              int converted\_number = htonl(message\_length);              write(client\_fd, &converted\_number, sizeof(converted\_number));              printf("\nMessage Length: %lld\n", message\_length);              results\_file = fopen("results.txt", "r");              // read results into a buffer from the file then send to client              while (fgets(results\_buffer, BUFFSIZE, results\_file) != NULL) {                  if ((write(client\_fd, results\_buffer, strlen(results\_buffer)))== -1)                  {                      // throw error if results could not be sent.                      fprintf(stderr, "ERROR: Failed to send the results to client.\n");                      close(client\_fd);                      break;                  }              } // data transferred              printf("Results sent to the client.\n\n");              pclose(fp);  // closing command file              fclose(results\_file); // closing results file          } // While loop end, connection ended          //Close Connection Socket          close(client\_fd);      } // While loop end, server Exits      // Flow never reaches here      close(socket\_fd);      return 0;  } //End of main |

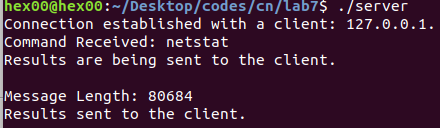
1-- netstat command with one argument



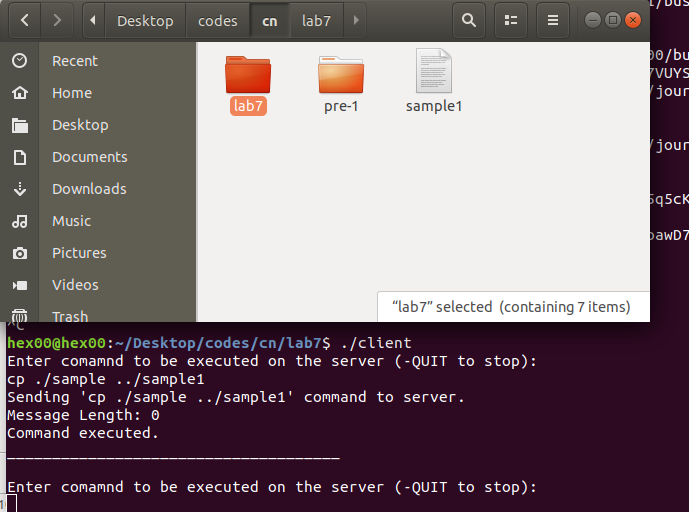
…



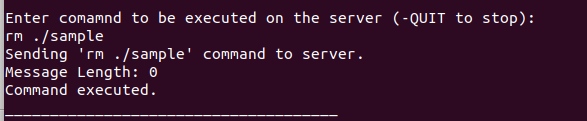
**Server Side:**



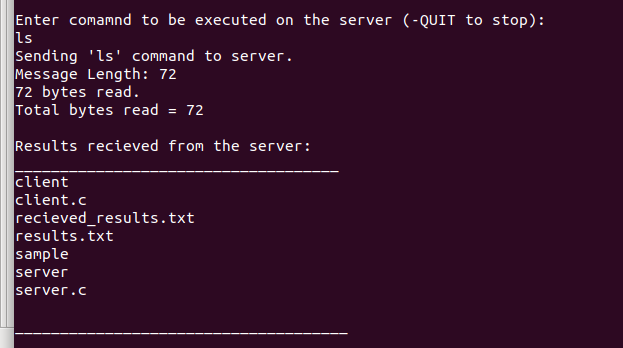
2-- cp command with two arguments



3-- rm command with one argument



4-- ls command without any argument



**Conclusion:**

In this lab, we learned and demonstrated the use of socket programming for Remote Procedure Calls. Conventionally, marshaling and stubs are used for RPC but other ways can also be used. For example, TCP can be used to transfer the commands in the form of packets from client to server. Server will execute the commands and return the results to the client.

**Deliverables:**

Objective, Source code with explanation, screenshots of output and a paragraph of Conclusion .