

RYERSON UNIVERSITY

Faculty of Engineering and Architectural Science

Department of Electrical and Computer Engineering



Program: Computer Engineering

Course Number	COE768
Course Title	Computer Networks
Semester/Year	Fall 2017
Instructor	Dr. Ngok – Wah Ma

Final Project Report

Report Title	File Transfer Application
--------------	---------------------------

Section No.	01 / 04
Submission Date	Wednesday, November 15, 2017
Due Date	Thursday, November 16, 2017

Name	Student ID	Signature*
Deon Botelho	500 576 967	
Vruti Vaghela	500 568 962	

**By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: <http://www.ryerson.ca/senate/policies/pol60.pdf>.*

Introduction:

The main objective of this project was to implement a file transfer system which will be referred to as a FTP throughout this report. The FTP system was implemented using Transmission Control Protocol (TCP). In order for the server to handle multiple file transfer requests the server must be a concurrent server. The FTP system consisted of uploading/downloading the files from the server, changing the working directory of the server, and list the current files the working server directory.

The TCP connection is a set of protocol in which the connection of network systems is done over the internet. In TCP, the messages which are to be transmitted/received are broken into smaller packets, which are assembled back to the whole message when the destination has been reached. TCP connections are used widely as it is a more reliable connection with a point-to-point communication channel. The TCP connection uses a client and server architecture.

Socket programming is a method in which two processes can communicate with one another on one network. These processes are more commonly known as the server and the client which build a connection to communicate with each other. The client and server processes both read and write to a socket which makes the communication possible. A socket is bound to the end of the connection for both the client and the server. The socket binding is key between the client and server connection. Each socket consists of an IP address and a port number on a system. The flow diagram below (Figure 1.) can be referred to see how sockets are connected between the client and the server architecture.

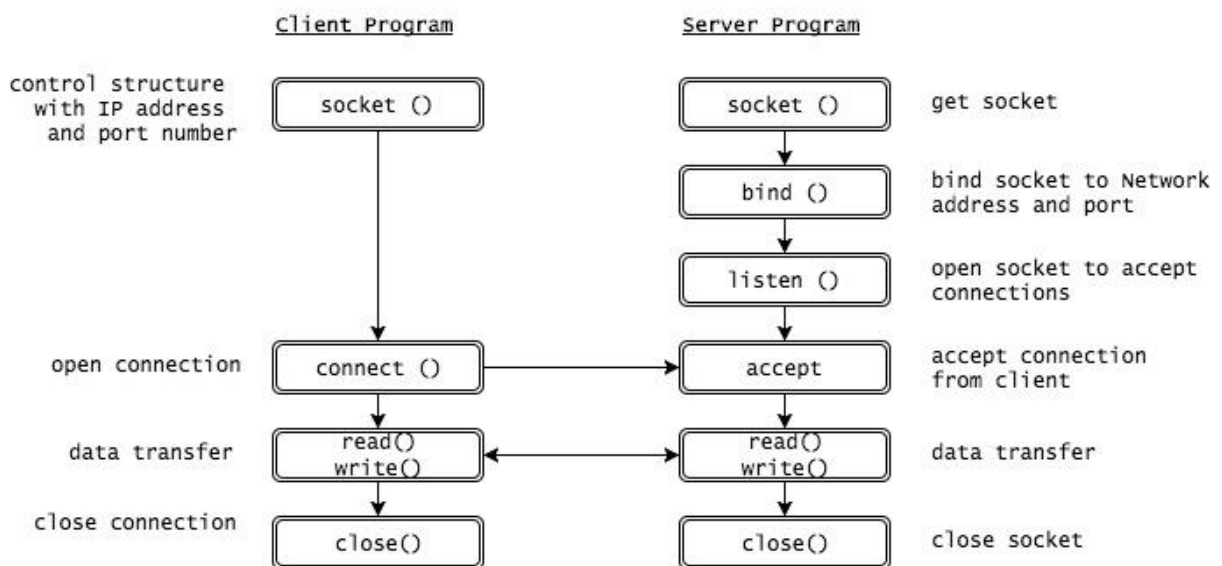


Figure 1: Flow Diagram of Socket Program

Description:

Client summary

Reads in user commands to specify the type of data transfer between itself and the server.

Functionality of the client

The main purpose of the client is to request data to be sent to (uploaded), or received from (downloaded) from the server. To do this various protocol data units (PDUs) are used to specify the data type and how the server (or client) should interpret the transferred data. These PDUs range from representing specific commands such as “download a file” or “upload a file”, or can represent the type of data to be sent or received. Since both the server and client must know what each PDU means there may be crossover in which PDUs both may be able to send.

Types of PDUs sent from the client and received from the server

Download request ('D'):

The download PDU is sent by the client to request a file to be download from the server. The filename is specified in the data portion of the data package and is interpreted as such by the server. Once the request has been sent, the client will go into a reading loop and continually write the incoming data from the server into a file with the specified name. This read/write loop is broken in only two scenarios. In the first case, the server has successfully finished sending all the data in the specified file and is ready to receive another command. As such the server will send the final data package with an EOT (end of transmission) PDU. In the second case, a PDU that is not expected, i.e. not a data PDU is received. In this case an error would have occurred and the client will close the write file and delete its existing contents since the data transfer was compromised. This will prompt the client to send an error transmission to the server to act accordingly and return to a ready state.

Upload request ('U'):

The upload PDU is sent by the client to request a file to be uploaded to the server. As in the download request, the file name is specified in the data portion of the data package. Once the request has been sent to the server the client will open the file to be transferred and again enter a read/write loop, this time to write to the server rather than to an actual file. The contents of the file are packaged in a PDU with type 'F' which stands for data. The read transfer loop will break only when the client has successfully finished sending the entire contents of the file or if an error has occurred. Once out of the read /write loop the client will return to its ready state awaiting a command from the user.

Data package ('F'):

The data PDU can be sent from either the client or server but in this section only the client side will be discussed. As mentioned before in the download and upload descriptions the data PDU is used when the type of data transfer has already occurred. For example, when the transfer type is specified to be an upload transaction, the data PDU is used for the file contents being sent from the client to the server. The data PDU is used only after the specific task has been established so that both the client and server know how to interpret the data.

Error ('E'):

The error PDU type is used by the client when an error has occurred in executing a particular task. This can range from a file failing to opening to an unexpected PDU being received. When an error occurs from the client side an error message is written in the data portion of the PDU and sent to the server for debugging reasons. If the error PDU is instead received from the server the client will process the error, printing out any error messages received and return to a ready state awaiting the next user command.

Change directory request ('P'):

The change directory PDU is sent by the client to the server to request a change in the current server directory. The new path is specified in the data portion of the PDU to be verified by the server. If successful the server will continue all transactions to occur at the specified path. If the given path does not exist or an error occurs the directory will not change and the client will be notified with an error PDU. The client may also return to the default directory by specifying the path as the value 'root', to which the server will return to the original directory.

List files request ('L'):

The list files request is sent by the client to the server to receive a list of all the files in the current working directory. The path by default is the root directory where the server project is stored but this path may be changed by using the "Change directory" command. Once the request has been made to the server the client enters a read loop similarly to the download function in which the loop only breaks if the transmission ends or if an error has occurred. The received data (the list of files) is specified by a unique "list files data" PDU (described in the server section) and is printed onto the client's terminal.

End of transmission ('Z'):

The end of transmission (EOT) PDU type is used at the end of a data transfer sequence to prompt the server to return to a ready state. If the user enters an exit command on the client side the EOT PDU is used to terminate the client-server connection. The client program will close the server connection and terminate as will the server's child process handling the client's requests.

Server summary

Creates a child process to handle any incoming client requests

Functionality of the server

The main purpose of the server is to create a child process that will handle any incoming request made by a client program that wishes to connect. This entails various jobs and data transactions such as downloading or uploading a file to and from the client. To handle these events various PDU's are used to effectively validate and interpret the data. Since both the server and client must know what each PDU means there may be crossover in which PDUs both may be able to send.

Types of PDUs sent and received by the server

Download request ('D'):

When the server receives a download request in it's ready state the server will move into a data transmission state. The server will first attempt to open the specified file if it exists. In the case that the file fails to open an error message is sent to the client and the server will return to a ready state. If the file is successfully opened the server will enter a read/write loop reading the file contents and transferring it to the client using the data 'D' PDU type. The read/write loop ends when the file is fully read (last contents will use the EOT PDU type) or if an error occurs. In both cases the server will return to a ready state awaiting another client request.

Upload request ('U'):

When the server receives an upload request from the client while in the ready state it will go into a data receive state. The server will open an file to be written into with the same name as specified by the data portion of the upload request. It will then enter a read/write loop, writing any contents received from the client into the opened file. This loop will terminate when an EOT PDU is received or an error occurs. If an error occurs the created file is closed and deleted. In both cases the server will return to a ready state awaiting another client request.

Ready ('R'):

The ready PDU is sent by the server to the client to state that it is ready to receive a request. The PDU is sent every time the server enters the ready state after finishing a request or processing an error. The client may only send a request after receiving the ready PDU. Failing to do so will result in an errored transaction between the two processes.

Data ('F'):

As mentioned in the client section the data PDU is used by both the client and server. In the server, it is primarily used to send data to the client during an download (to client) transaction

and received when the client wishes to upload a file to the server. This PDU type can only be used once a transaction job has been established. An error message will be sent in any other event.

Error ('E')

The error PDU type is used by the server when an error has occurred in executing a particular task. This can range from a file failing to opening to an unexpected PDU being received. When an error occurs from the server side an error message is written in the data portion of the PDU and sent to the client for debugging reasons. If the error PDU is instead received from the client the server will process the error, printing out any error messages received and return to a ready state awaiting the next client request.

Change directory request ('P'):

The change in directory request PDU is received by server from the client. Once registered the server will verify that the given path (stored in the data portion of the PDU) is valid. If successfully verified the server will assume all future incoming transactions should occur in the newly specified path. The path may be changed again by the client returning back to the default directory by using the value 'root' instead of a specific path. If the given path fails to validate no change to the current directory will be made.

List files request ('L')

When the list files request is received by the server, the server will execute the terminal "ls" command and store the output contents into a temporary file. This file is then read and written to the client to be viewed by the user. After completing the write process, the temporary file will be deleted. If a file currently exists with the temporary files name it will be overwritten by the server.

List files data ('1')

The list files data PDU is a special data PDU used only in the case of a list files request. Only the server may use this PDU to relay the current files available in the working directory to the client. When received by the client the contents of the data will be directly output for the user to see unlike in the download and upload scenario where the contents are written to file to be view later.

End of transmission ('Z'):

The end of transmission (EOT) PDU type is used at the end of a data transfer sequence to prompt the client that the server has finished its job. When received instead of a specific job request such as download, the server will close the client socket and terminate the child process assigned to handle the client's request. The main server process however will remain running to handle any new incoming client connections that may be made in the future.

Observation and Analysis:

Download

The testing of the “download” command as requested by the client to the server can be seen below in Figure 2. Terminal one (top left) shows the server and its current output in response to the clients download request as shown in terminal two (top right). The user inputs the download command by entering “download data.txt” into the terminal. The upload request is invoked by the upload command “upload client_newdata.txt”. In response the server (terminal one) prints the whole path of the file to be uploaded to the client. The results can be view by looking at terminal three (bottom left) and four (bottom right) which show the current files in the working before and after the download command is invoked. Comparing the results of these two terminals a new file appears in the directory called “client_newdata.txt”. It should be noted that the prefix “client_new” has been concatenated to the filename for the purposes of distinguishing between the original and new data file created after the download transaction but the file contents remain completely identical. In this case because the client is written data from the server into a new file the prefix has been denoted as “client_new”.

```
echo_server
Himmats-MacBook-Pro:project1 vruti$ ./echo_server 5000

New Client 4 connected to :
/Users/vruti/Desktop/coe768/project1

Server connected: Ready sent
opening : /Users/vruti/Desktop/coe768/project1/data.txt
Server connected: Ready sent
[]

echo_client
Himmats-MacBook-Pro:project1 vruti$ ./echo_client localhost 5000
List of commands :
Download a file from server : "download [filename]"
Upload a file to the server : "upload [filename]"
Change directory in server : "directory [path]"
Note: To return to the root directory use [path] = root
List all files in directory : "listfiles"
List of available commands : "help"
Terminate service : "exit"
recv >>[R][S][ready]
Enter a command :
download data.txt
in download
finished
recv >>[R][S][ready]
Enter a command :
[]

bash
Himmats-MacBook-Pro:project1 vruti$ ls
data.txt  echo_client  echo_client.c  echo_server  echo_server.c
Himmats-MacBook-Pro:project1 vruti$

bash
Himmats-MacBook-Pro:project1 vruti$ ls
client_newdata.txt  echo_client  echo_server
data.txt           echo_client.c  echo_server.c
Himmats-MacBook-Pro:project1 vruti$
```

Figure 2: Download Command

Upload

The testing of the “upload” command as requested by the client to the server can be seen below in Figure 3. Terminal one (top left) shows the output of the server in response to the clients upload request as seen in terminal two (top right). The upload request is invoked by the upload command “upload client_newdata.txt”. In response, the server (terminal one) prints the whole path of the file to be downloaded from the client. The results of the upload transaction can be view in terminal three (bottom left) and terminal four (bottom right) which show the before and after of when the upload command is called. Comparing these two terminals a new file called “server_newclient_newdata.txt” appears after the upload command had been invoked. As in the download example the prefix “server_new” portion of the file name has been concatenated on to distinguish between the original and the new file created after the transaction. In this case because it was the server receiving the file, the prefix is written on the server side, hence “server_new”.

```
echo_server
Himmats-MacBook-Pro:project1 vruti$ ./echo_server 5001

New Client 4 connected to :
/Users/vruti/Desktop/coe768/project1

Server connected: Ready sent
opening : /Users/vruti/Desktop/coe768/project1/server_newclient_newdata.txt
Server connected: Ready sent
[]

echo_client
Himmats-MacBook-Pro:project1 vruti$ ./echo_client localhost 5001

List of commands :
Download a file from server : "download [filename]"
Upload a file to the server : "upload [filename]"
Change directory in server : "directory [path]"
Note: To return to the root directory use [path] = root
List all files in directory : "listfiles"
List of available commands : "help"
Terminate service : "exit"
recv >>[R][S][ready]
Enter a command :
upload client_newdata.txt
in upload
finished
recv >>[R][S][ready]
Enter a command :
[]

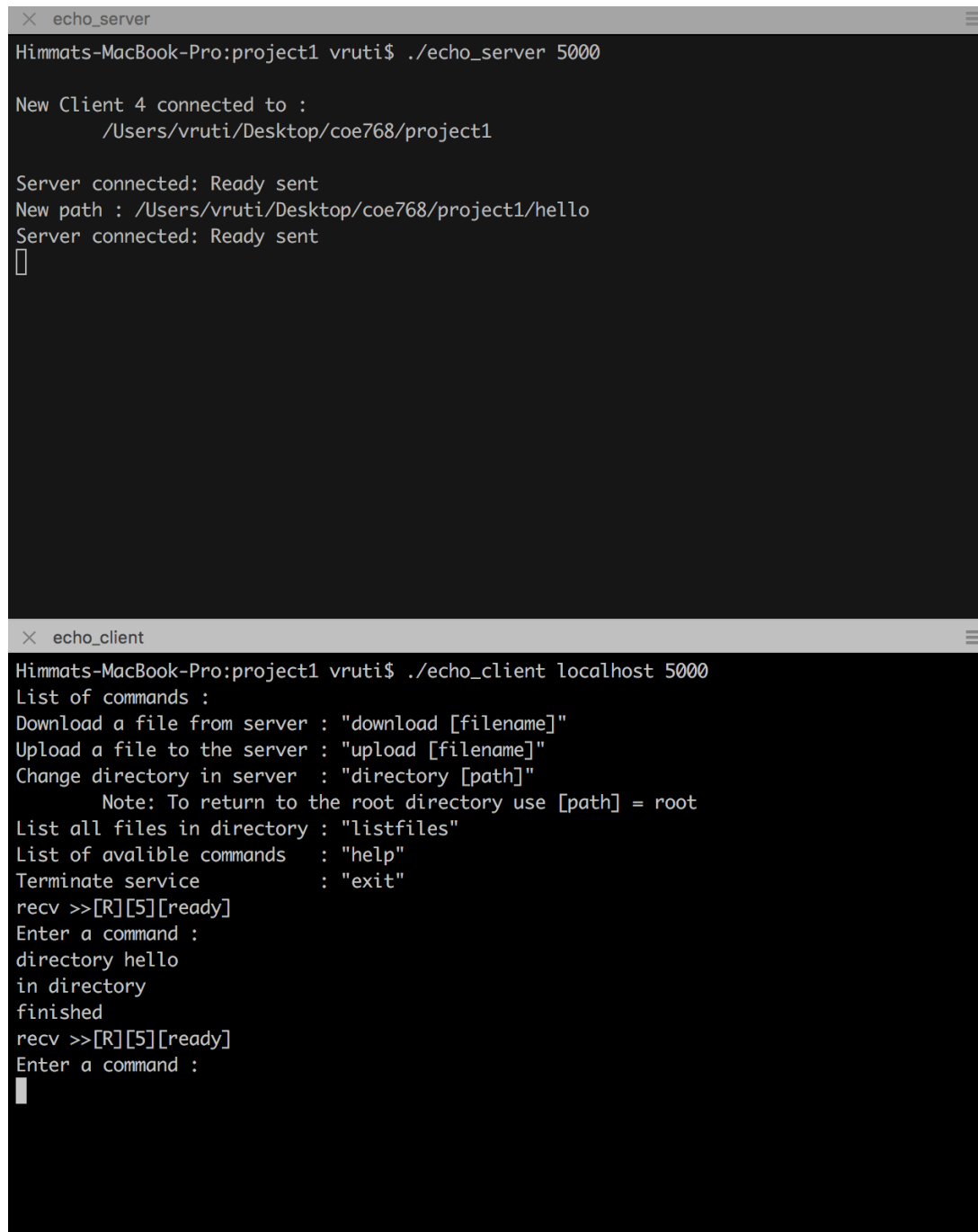
bash
Himmats-MacBook-Pro:project1 vruti$ ls
client_newdata.txt  echo_client
data.txt            echo_client.c
Himmats-MacBook-Pro:project1 vruti$

bash
Himmats-MacBook-Pro:project1 vruti$ ls
client_newdata.txt  echo_server
data.txt            echo_server.c
echo_client         server_newclient_newdata.txt
echo_client.c
Himmats-MacBook-Pro:project1 vruti$
```

Figure 3 Upload Command

Change directory:

The testing of the “change directory” command can be seen in Figure 4. The output of the server can be seen in terminal one (top) which shows the change in the working directory when the path has been changed by the client. Terminal two (bottom) shows the change directory command being invoked by the command “directory hello” which changes the current working directory to the folder “hello” currently in the shared folder.



```
echo_server
Himmats-MacBook-Pro:project1 vruti$ ./echo_server 5000

New Client 4 connected to :
    /Users/vruti/Desktop/coe768/project1

Server connected: Ready sent
New path : /Users/vruti/Desktop/coe768/project1/hello
Server connected: Ready sent
█

echo_client
Himmats-MacBook-Pro:project1 vruti$ ./echo_client localhost 5000
List of commands :
Download a file from server : "download [filename]"
Upload a file to the server : "upload [filename]"
Change directory in server : "directory [path]"
    Note: To return to the root directory use [path] = root
List all files in directory : "listfiles"
List of available commands : "help"
Terminate service : "exit"
recv >>[R][5][ready]
Enter a command :
directory hello
in directory
finished
recv >>[R][5][ready]
Enter a command :
█
```

Figure 4 Change Directory Command

List Files

The testing of the “list files” command is shown below in Figure 5. The output of the server can be seen in terminal one (top) which shows the current working directory. The list files command is invoked by the client as seen in terminal two (bottom) by the command “listfiles”. The server prints out the list of files that is to also be sent to the client which in turn also prints out the results of the transaction. When comparing the two lists in both terminals a duplicate list can be seen proving that the transaction was successful.

```
C:\Users\Deon\Desktop\COE768\FileTransferProject\server>echo_server 500
New Client 4 connected to :
    /cygdrive/c/Users/Deon/Desktop/COE768/FileTransferProject/server
Server connected: Ready sent
in listfiles: root
/cygdrive/c/Users/Deon/Desktop/COE768/FileTransferProject/server
ls -Q>> tmp_ls_file.txt
tmp file open
"data.txt"
"echo_server.c"
"echo_server.exe"
"hello"
"tmp_ls_file.txt"
[Z]doneServer connected: Ready sent
```

```
C:\Users\Deon\Desktop\COE768\FileTransferProject\client>echo_client 10.16.187.111 500
List of commands :
Download a file from server : "download [filename]"
Upload a file to the server : "upload [filename]"
Change directory in server : "directory [path]"
    Note: To return to the root directory use [path] = root
List all files in directory : "listfiles"
List of available commands : "help"
Terminate service : "exit"
recv >>[R][5][ready]
Enter a command :
listfiles
in listfiles
[Z]
"data.txt"
"echo_server.c"
"echo_server.exe"
"hello"
"tmp_ls_file.txt"
Finished Listing files

finished
recv >>[R][5][ready]
Enter a command :
```

Figure 5 List Files Command

Conclusion:

In conclusion, the FTP project was a successful in using all required PDU types specified in the project guidelines. Each PDU type was implemented with a particular task or significance behind it and was simplified by creating simple commands in the client application for the user to use. These commands in turn used the PDUs to signify the type of data being transferred over the TCP connection so that both the client and server could accurately interpret the data being read. All significant tasks have been tested and proven to work through demonstration in the lab as well as in the observations of this lab report.

Appendix:

Server Program:

```
/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <strings.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <netdb.h>
#include <sys/signal.h>
#include <sys/wait.h>

#define SERVER_TCP_PORT 3000 /* well-known port */
#define BUFLLEN 512 /* buffer length */
#define DATALEN 100

int echod(int);
void reaper(int);
int uploadFiletoClient(int sd,char *fileName,int strlength);
int downloadFileFromClient(int sd, char *fileName, int strlength);
void sendReady(int sd);
void sendErrorMsg(int sd, char *errorMsg, int strlength);
int changeDirectory(char *path, int strlength);
int listFiles(int sd);

char homopath[BUFLLEN], clientpath[BUFLLEN],subpath[BUFLLEN] ="root";
struct PDUData
{
    char type;
    int length;
    char data[DATALEN];
} receivedData, transmittedData;
struct PDUType {
    char download;//D:client>>server request to download specified file
    char upload; //U:client>>server request to upload specified file
    char ready; //R:server>>client ready to recieved a request from client
    char data; //F:client<>server response data sent and received
    char error; //E:client<>server error has occurred
    char cd; //P:client>>server request to change directory
    char ls; //L:client>>server request a list of all files in current directory
    char lsData; //1:server>>client response to request for list of files
    char EOT; //Z:client<>server end of transmission
}PDU = { 'D','U','R','F','E','P','L','1','Z'};
```

```

int main(int argc, char **argv) {
    int sd, new_sd, client_len, port;
    struct sockaddr_in server, client;

    switch (argc) {
    case 1:
        port = SERVER_TCP_PORT;
        break;
    case 2:
        port = atoi(argv[1]);
        break;
    default:
        fprintf(stderr, "Usage: %d [port]\n", argv[0]);
        exit(1);
    }

    /* Create a stream socket */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't creat a socket\n");
        exit(1);
    }

    /* Bind an address to the socket */
    bzero((char *) &server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(sd, (struct sockaddr *) &server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }

    /* queue up to 5 connect requests */
    listen(sd, 5);

    (void) signal(SIGCHLD, reaper);

    while (1) {
        client_len = sizeof(client);
        new_sd = accept(sd, (struct sockaddr *) &client, &client_len);
        if (new_sd < 0) {
            fprintf(stderr, "Can't accept client \n");
            exit(1);
        }
        switch (fork()) {
        case 0: /* child */
            (void) close(sd);
            exit(echod(new_sd));
        default: /* parent */
            (void) close(new_sd);
            break;
        case -1:
            fprintf(stderr, "fork: error\n");
        }
    }
}

```

```

/*      echod program      */
int echod(int sd) {
    int n, valid;
    getcwd(homepath, BUFLLEN);
    strcpy(clientpath, homepath);
    printf("\nNew Client %d connected to : \n\t%s\n",sd, clientpath);
    for (;;) {
        printf("Server connected: ");
        sendReady(sd);
        printf("Ready sent\n");
        n = read(sd,&receivedData, sizeof(struct PDUDData)); //receive cmd from server
        if (receivedData.type == PDU.download)
        {
            valid = uploadFiletoClient(sd, receivedData.data, receivedData.length);
        }
        else if (receivedData.type == PDU.upload)
        {
            valid = downloadFileFromClient(sd, receivedData.data, receivedData.length);
        }
        else if (receivedData.type == PDU.cd)
        {
            valid = changeDirectory(receivedData.data, receivedData.length);
        }
        else if (receivedData.type == PDU.ls)
        {
            valid = listFiles(sd);
        }
        else if (receivedData.type == PDU.EOT)
        {
            break;
        }
        else if (receivedData.type == PDU.error)
        {
            printf("error recived :%s\n",receivedData.data);
        }
        else //error
        {
            valid = -1;
        }
        if (valid<0)//valid < 0
        {
            printf("error\n");
            sendErrorMsg(sd, "Error processing request", strlen("Error processing request"));
        }
    }
    close(sd);
    return 0;
}

/*      reaper      */
void reaper(int sig) {
    int status;
    while (wait3(&status, WNOHANG, (struct rusage *) 0) >= 0);
}

```

```

/*Called if client wishes to download a file from the server*/
int uploadFiletoClient(int sd, char *fileName, int strlength)
{
    int n, readFile;// = open(fileName, O_RDONLY);
    char filepath[BUFLen];
    sprintf(filepath, "%s/%s", clientpath, fileName);
    readFile = open(filepath, O_RDONLY);
    if (readFile > 0)
    {
        printf("opening : %s\n ", filepath);
        while ((n = read(readFile, transmittedData.data, DATALEN)) > 0)
        {
            transmittedData.type = n==DATALEN?PDU.data:PDU.EOT;
            transmittedData.length = n;
            write(sd, &transmittedData, sizeof(transmittedData));
            if (transmittedData.type == PDU.EOT)
                break;
        }
        close(readFile);
    }
    return readFile;
}

/*Called if the client wishes to upload to the server*/
int downloadFileFromClient(int sd, char *fileName, int strlength)
{
    char filepath[BUFLen];
    int n, writeFile;

    sprintf(filepath, "%s/%s%s", clientpath, "server_new", fileName);
    writeFile = open(filepath, O_WRONLY | O_CREAT, S_IRWXU);
    if (writeFile > 0)
    {
        printf("opening : %s\n ", filepath);
        while ((n = read(sd, &receivedData, sizeof(struct PDUData))) > 0)
        {
            if (receivedData.type == PDU.data || receivedData.type == PDU.EOT)
            {
                write(writeFile, receivedData.data, receivedData.length);
                if (receivedData.type == PDU.EOT)
                    break;
            }
            else
            {
                close(writeFile);
                sprintf(receivedData.data, "%s %s", "rm", filepath);
                system(receivedData.data);
                return -1;
            }
        }
        close(writeFile);
    }
    return writeFile;
}

```

```

/*Called to tell client,server is ready for a cmd*/
void sendReady(int sd)
{
    transmittedData.type = PDU.ready;
    strcpy(transmittedData.data, "ready");
    transmittedData.length = strlen(transmittedData.data);

    write(sd, &transmittedData, sizeof(transmittedData));
}

/*Called if any error has occurred*/
void sendErrorMsg(int sd, char *errorMsg, int strlength)
{
    transmittedData.type = PDU.error;
    strcpy(transmittedData.data, errorMsg);
    transmittedData.length = strlen(transmittedData.data);

    write(sd, &transmittedData, sizeof(transmittedData));
}

/*Called if the client wishes to change the current directory*/
int changeDirectory(char *path, int strlength)
{
    char cmd[BUFLEN] = "cd ";
    if (strcmp(path, "root") == 0)
    {
        strcpy(clientpath, homedir);
        strcpy(subpath, path);
        printf("New path : %s\n", clientpath);
        return 0;
    }
    else
    {
        strcat(cmd, path);
        if (system(cmd) == 0) //is the path valid?
        {
            sprintf(clientpath, "%s/%s", clientpath, path);
            strcpy(subpath, path);
            printf("New path : %s\n", clientpath);
            return 0;
        }
    }
    return -1;
}

/*Called if the client wishes to know what is in the current directory*/
int listFiles(int sd)
{
    char cmd[BUFLEN], tmp[BUFLEN];
    int tmpFile, valid, n;
    printf("in listfiles: %s\n%s\n", subpath, clientpath);
    if (strcmp(subpath, "root") != 0)
    {
        sprintf(cmd, "%s %s %s", "cd", subpath, "&& ls -Q>> tmp_ls_file.txt");
    }
}

```



```

else
{
    strcpy(cmd, "ls -Q>> tmp_ls_file.txt");
}

printf("%s\n", cmd);
valid = system(cmd);
sprintf(tmp, "%s/%s", clientpath, "tmp_ls_file.txt");
tmpFile = open(tmp, O_RDONLY);

if (valid >= 0 && tmpFile > 0)
{
    printf("tmp file open\n");
    while ((n = read(tmpFile, transmittedData.data, DATALEN))>=0)
    {
        transmittedData.type = n == DATALEN ? PDU.lsData : PDU.EOT;
        transmittedData.length = n;
        write(sd, &transmittedData, sizeof(transmittedData));
        printf("[%c]", transmittedData.type);
        write(1, transmittedData.data, transmittedData.length);
        if (transmittedData.type == PDU.EOT)
        {
            printf("done");
            close(tmpFile);
            break;
        }
    }
}

if (strcmp(subpath, "root") != 0)
{
    sprintf(cmd, "%s %s %s", "cd", subpath, "&& rm tmp_ls_file.txt");
}
else
{
    strcpy(cmd, "rm tmp_ls_file.txt");
}
system(cmd);
return valid;
}

```

Client Program:

```
/* A simple echo client using TCP */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

#define SERVER_TCP_PORT 3000 /* well-known port */
#define BUFLLEN 512 /* buffer length */
#define DATALEN 100

int downloadFileFromServer(int sd,char *fileName,int strlength);
int uploadFileToServer(int sd, char *fileName, int strlength);
int changeDirectory(int sd, char *path, int strlength);
int listFiles(int sd);
int help();
int sendErrorMsg(int sd, char *errorMsg, int strlength);

struct PDUData {
    char type;
    int length;
    char data[DATALEN];
} receivedData, transmittedData;

struct PDUType {
    char download;//D:client>>server request to download specified file
    char upload; //U:client>>server request to upload specified file
    char ready; //R:server>>client ready to recieved a request from client
    char data; //F:client<>server response data sent and received
    char error; //E:client<>server error has occurred
    char cd; //P:client>>server request to change directory
    char ls; //L:client>>server request a list of all files in current directory
    char lsData; //1:server>>client response to request for list of files
    char EOT; //Z:client<>server end of transmission
}PDU = { 'D','U','R','F','E','P','L','1','Z' };
```

```

int main(int argc, char **argv) {
    int n, i, newFile, valid;
    int sd, port;
    struct hostent *hp;
    struct sockaddr_in server;
    char *host, tmp[DATALEN], serverDir[DATALEN], cmd[DATALEN], *pt;

    switch (argc) {
    case 2:
        host = argv[1];
        port = SERVER_TCP_PORT;
        break;
    case 3:
        host = argv[1];
        port = atoi(argv[2]);
        break;
    default:
        fprintf(stderr, "Usage: %s host [port]\n", argv[0]);
        exit(1);
    }

    /* Create a stream socket */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't creat a socket\n");
        exit(1);
    }

    bzero((char *)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    server.sin_port = htons(port);
    if (hp = gethostbyname(host))
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
    else if (inet_aton(host, (struct in_addr *)&server.sin_addr)) {
        fprintf(stderr, "Can't get server's address\n");
        exit(1);
    }

    /* Connecting to the server */
    if (connect(sd, (struct sockaddr *)&server, sizeof(server)) == -1) {
        fprintf(stderr, "Can't connect \n");
        exit(1);
    }

    help();
    for (;;)
    {
        /* Client loop : Wait for command */
        read(sd, &receivedData, sizeof(struct PDUData));
        printf("recv >>[%c][%d][%s]\n", receivedData.type, receivedData.length, receivedData.data);
        if (receivedData.type == PDU.ready)
        {
            printf("Enter a command : \n");
            n = read(0, cmd, DATALEN);

```

```

for (i = 0; i < DATALEN; i++)
{
    if (cmd[i] == ' ')
    {
        cmd[i] = '\0';
        i++;
        break;
    }
}
cmd[n - 1] = 0;
pt = &cmd[i];
strcpy(tmp, pt);
//printf("<%s> : <%s> : <%s>\n", cmd, tmp, pt);

if (strcmp(cmd, "download") == 0)
{
    printf("in %s\n", cmd);
    valid = downloadFileFromServer(sd, tmp, n - i);
}
else if (strcmp(cmd, "upload") == 0)
{
    printf("in %s\n", cmd);
    valid = uploadFileToServer(sd, tmp, n - i);
}
else if (strcmp(cmd, "directory") == 0)
{
    printf("in %s\n", cmd);
    valid = changeDirectory(sd, tmp, n - i);
}
else if (strcmp(cmd, "listfiles") == 0)
{
    printf("in %s\n", cmd);
    valid = listFiles(sd);
}
else if (strcmp(cmd, "help") == 0)
{
    printf("in help\n", cmd);
    sendErrorMsg(sd, "H", 1);
    valid = 100;
    help();
}
else if (strcmp(cmd, "exit") == 0)
{
    printf("in %s\n", cmd);
    transmitedData.type = PDU.EOT;
    transmitedData.data[0] = '\0';
    transmitedData.length = 0;
    write(sd, &transmitedData, sizeof(transmitedData));
    exit(0);
}
else
{
    printf("Invalid command! Type help for a list of valid commands.\n");
}

```

```

        }

    }
    else
    {
        valid = -1;
    }

    if (valid < 0)
    {
        printf("in error\n");
        sendErrorMsg(sd, "Error receiving", strlen("Error receiving"));
    }
    printf("finished\n");
}

close(sd);
return (0);
}

/*Request to download a specific file from the server*/
int downloadFileFromServer(int sd,char *fileName,int strlength)
{
    char tmp[DATALEN] = "client_new";
    int n, newfile;

    strcat(tmp, fileName);
    newfile= open(tmp, O_WRONLY | O_CREAT, S_IRWXU);

    if (newfile > 0)
    {
        transmittedData.type = PDU.download;
        strcpy(transmittedData.data, fileName);
        transmittedData.length = strlength;

        write(sd, &transmittedData, sizeof(transmittedData)); //send req

        while ((n = read(sd, &receivedData, sizeof(struct PDUData))))
        {
            if (receivedData.type == PDU.data || receivedData.type == PDU.EOT)
            {
                write(newfile, receivedData.data, receivedData.length);
                if (receivedData.type == PDU.EOT)
                {
                    close(newfile);
                    return 0;
                }
            }
            else
            {
                close(newfile);
                sprintf(receivedData.data, "%s %s", "rm", tmp);
            }
        }
    }
}

```

```

        printf("error:%s\n",receivedData.data);
        system(receivedData.data);
        return -1;
    }
}
return -1;
}
/*Request to upload a specific file from the server*/
int uploadFileToServer(int sd, char *fileName, int strlength)
{
    int n, readFile = open(fileName, O_RDONLY);
    if (readFile > 0)
    {
        transmittedData.type = PDU.upload;
        strcpy(transmittedData.data, fileName);
        transmittedData.length = strlength;

        write(sd, &transmittedData, sizeof(transmittedData)); //send req

        while ((n = read(readFile, transmittedData.data, DATALEN)) > 0)
        {
            transmittedData.type = n == DATALEN ? PDU.data : PDU.EOT;
            transmittedData.length = n;
            write(sd, &transmittedData, sizeof(transmittedData));
            if (transmittedData.type == PDU.EOT)
                break;
        }
        close(readFile);
    }
    return readFile;
}

int changeDirectory(int sd, char *path, int strlength)
{
    transmittedData.type = PDU.cd;
    strcpy(transmittedData.data, path);
    transmittedData.length = strlength;

    write(sd, &transmittedData, sizeof(transmittedData));
    return 0;
}

int listFiles(int sd)
{
    transmittedData.type = PDU.ls;
    transmittedData.data[0] = 0;
    transmittedData.length = 0;
    write(sd, &transmittedData, sizeof(transmittedData));
    while ( ( read(sd, &receivedData, sizeof(struct PDUData)))>0)
    {
        printf("[%c]\n", receivedData.type);
        if (receivedData.type == PDU.lsData || receivedData.type == PDU.EOT)

```

```

        {
            write(1, receivedData.data, receivedData.length);
            if (receivedData.type == PDU.EOT)
            {
                break;
            }
        }
        else
        {
            return -1;
        }
    }
    printf("Finished Listing files \n\n");
    return 0;
}

```

```

int help() {
    printf("List of commands : \n");
    printf("Download a file from server : \"download [filename]\" \n");
    printf("Upload a file to the server : \"upload [filename]\" \n");
    printf("Change directory in server : \"directory [path]\" \n");
    printf("\tNote: To return to the root directory use [path] = root\n");
    printf("List all files in directory : \"listfiles\" \n");
    printf("List of available commands : \"help\" \n");
    printf("Terminate service : \"exit\" \n");
    return 0;
}

```

/*Called if any error has occurred*/

```

int sendErrorMsg(int sd, char *errorMsg, int strlength)
{
    printf("%s\n", errorMsg);
    transmittedData.type = PDU.error;
    strcpy(transmittedData.data, errorMsg);
    transmittedData.length = strlen(transmittedData.data);
    printf("[%c][%d][%s]\n", transmittedData.type, transmittedData.length, transmittedData.data);
    write(sd, &transmittedData, sizeof(transmittedData));
    return 0;
}

```