




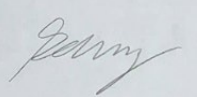
**Department of Electrical
& Computer Engineering**
Faculty of Engineering & Architectural Science

Course Title:	Computer Networks
Course Number:	COE768
Semester/Year (e.g.F2016)	F2019

Instructor:	Tasquia Mizan
--------------------	----------------------

<i>Assignment/Lab Number:</i>	Project
<i>Assignment/Lab Title:</i>	File Transfer

<i>Submission Date:</i>	Nov. 15 th , 2019
<i>Due Date:</i>	Nov. 15 th , 2019

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Chen	Jie	500715242	2	
Zou	Yun Peng	500628304	2	

Reset Form

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report 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/current/pol60.pdf>

Table of Contents

1.0 – Introduction.....	2
2.0 – Detailed Description.....	3
3.0 – Observation.....	5
4.0 – Analysis.....	8
5.0 – Conclusion.....	9
6.0 – Appendix.....	10

1.0 - Introduction

In the modern era, an integral part of networking is to be able to remotely access the files. This project will explore the concept of transferring of file transferring. The program will be able to upload from client to server and download from server to client. The location of the download will not change and will be the executable file directory. The location of upload can be changed with command and the default directory is the executable file current directory. An upload and download is added to the end of file name after transferred to make the program differentiate between downloaded files and uploaded files in the case they are both targeting the same directory. These were achieved through socket programming.

1.1 - Socket Programming Basics

Socket programming connects two nodes on a network to communicate to one another. The socket listens to the port at an IP. While the other socket reaches out to form a connection.

State diagram for server and client mode is showed in Figure 1 (<https://www.geeksforgeeks.org/socket-programming-cc/>)

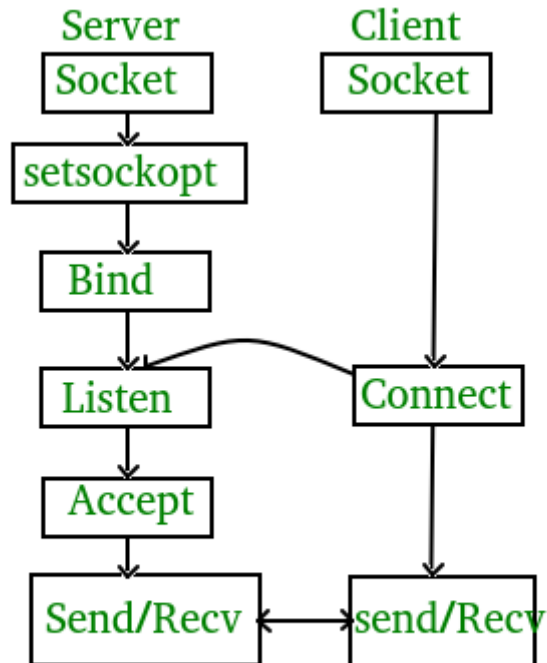


Figure 1

2.0 - Detailed Description

Since this is based on TCP protocol, and as learned from Lab 5, the basic approach of the client and server program is by setting two PDU data structures *tpdu* and *rpdu* that contains three fields for the type of the PDU, the size of the data and the value of the data. These two structures are used to send signals and data between the client and server programs. When ever a client reads an instruction entered by the user, it sends the type of the instruction along with the data (stored in the PDU) to the server. After receiving the PDU, the server reads the type and data, and performs the assigned operations for the specified type of instruction, and re-send the PDU back to the client with different information regarding about the state and process of the operation. Finally, the client reads the PDU sent by the server, makes sure that everything is settled, then prints out the results and acknowledges the user about the outcome of the assigned instruction.

There are four types of instructions for users to pick, which are the type 'D' for file downloads, where the client requests a file to download from the current working directory with the specific file name by the user; and the type 'U' for file uploads, where the client requests a file upload to the current working directory with the existing file. Besides these four types of PDU, there are other four types of PDU used during a request or instruction. The 'R' type is sent by the server to indicate that the program is ready, therefore the client can start executing the request; the 'F' type is used when a file download or upload request is received, it contains the data and the size of the file; the 'E' type indicates that there is an error during the progress, which is detected and sent by the server; and last the 'l' type is sent by the server where it contains the list of file names.

For the client program, the implementation lies under the main program (`int main (...)`) after the section "connecting to the server" where it successfully connects with the server. There is a loop after the connection is successful where it reads the input data from the user and sets up four switch cases for different inputs. The server program on the other hand, the implementation happens under the *echod* program (`int echod (int sd)`), and similarly with a loop and four switch cases. Each case corresponds to the type of the PDU, indicated by the corresponding letter representations.

For case 'D', the client reads the name of the file and store it to the PDU and notify the server. After server receives the PDU, it checks if the file exists or not by opening it. If the file does not exist, it will return an error PDU back to the client. If file exists, then it reads the data in the file and store into the PDU and set the type to 'F'. When client receives the PDU sent from the server, it checks if the PDU is 'F', otherwise it prints out an error message on the command prompt. If the PDU is type 'F', then it will create a new file with the same file name to be downloaded with 'download' at the end of the name. It then reads the file data stored within the PDU and writes it into the new file that is created. This is similar for case 'U', the only difference is that the server will first set the PDU type to 'R', indicating that the new file is created and opened, and ready to receive the data packets.

For case 'P', the client reads the path entered by the user and stores it into the PDU and send to the server. When server receives the PDU, it will first check the validity of the path by opening the directory and also changing the directory in the meantime. If the path is valid, it will write the type 'R' into the PDU and send to the client, otherwise it will return an error type 'E' PDU. After the 'R' type PDU is sent by the server, the client then breaks the case and prints out the results on the command prompt. For case 'L', it is similar to the case 'P' where it reads the path and stores into the PDU and send to the server. When the server receives it, it will also check the validity of the path, and then open the path and read the contents. It reads all the names of the files and store them into a string by detecting the NULL value. Whenever it reads a NULL letter, it will put the next file name into the string. After reading and storing all the files, it will store the string into the PDU and sent to the client. It will also return an error PDU if the path is not valid. When client receives the list, it will just print them out and break the case.

We added an extra case 'E' for the client program as an interesting feature, where the client can just send an error message to the server, and the server will return back the error PDU back to the client and prints out "Invalid pdu" on the prompt, and the server would also prints out the same text on the prompt.

3.0 - Observation

When the server and client are first created the client will display a prompt. In the beginning, if you download and upload file will be in the same directory if you have it in the same folder like in **Figure 2**. To differentiate between downloading and uploading was added to the end of the name which could be easily changed if you remove the code string concatenate command. After the download/upload command, you will need to enter the file name and you need to include the file extension or it will have problems finding it. The list directory will show all files in server's current directory. For change directory, the program requires the user to enter an absolute path for the new directory. To get an absolute directory, type *pwd* on terminal at the directory you want to use. **Figures 2 to 4** show the print screen of the command prompt and the results of each request. The server side was not included since it only prints out a prompt when each command is complete.

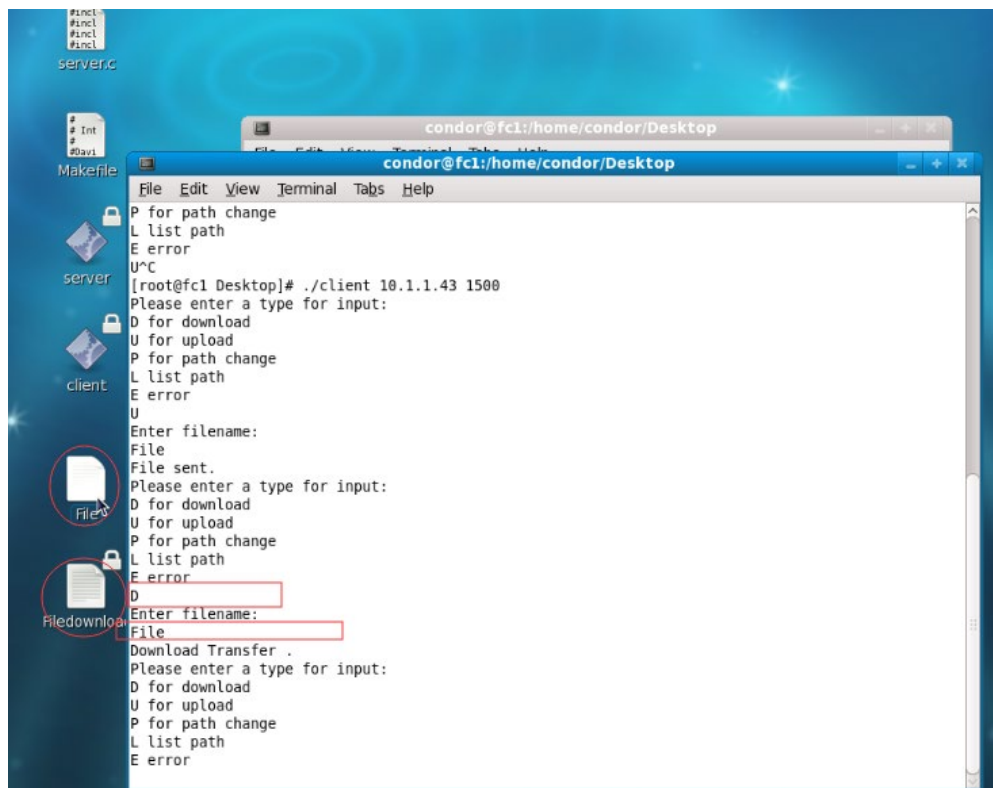


Figure 2: Downloading the Files

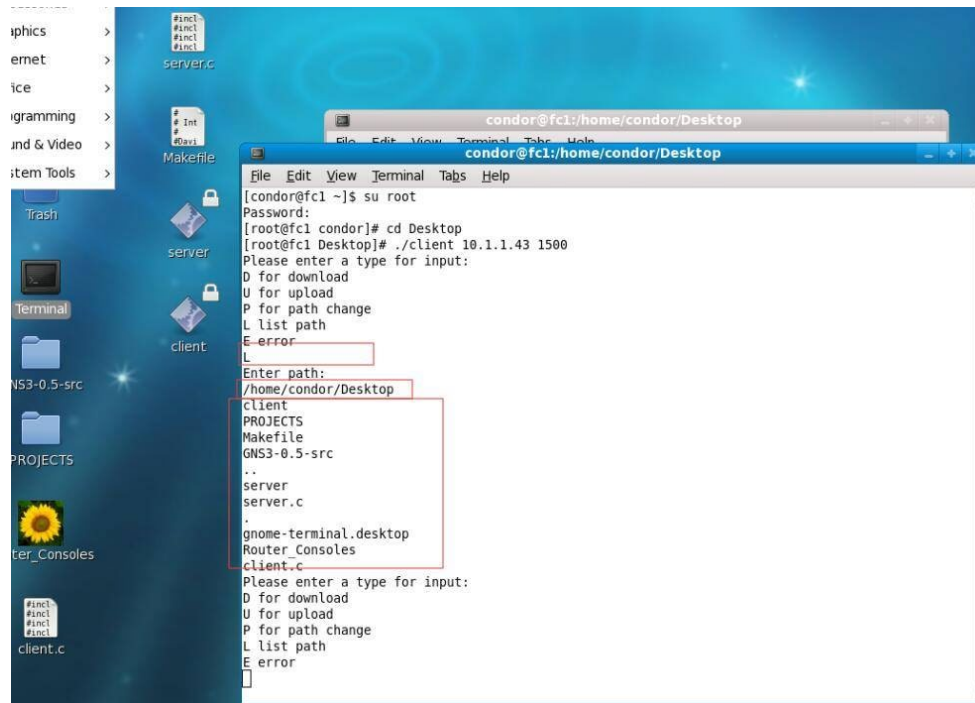


Figure 3: List folders in current directory of the server

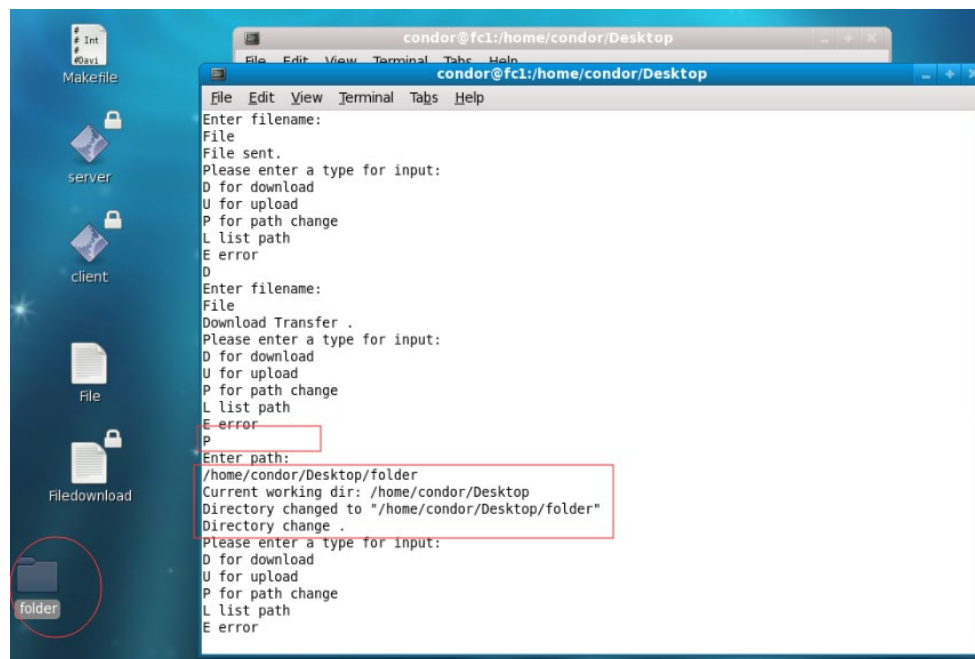


Figure 4: Change the current directory of the server

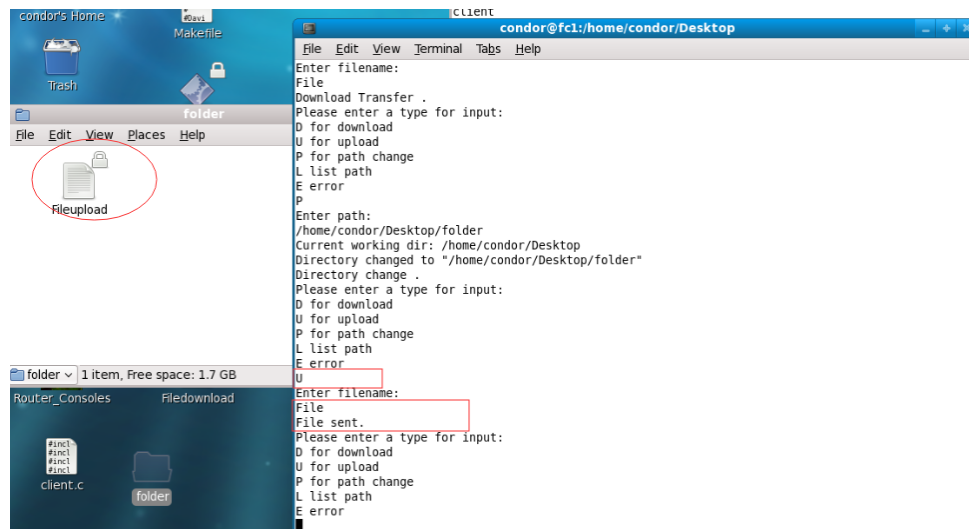


Figure 5: Upload the current directory of the server

4.0 - Analysis

The program works according to the specification. The program downloads and uploads to base upon the current directory of the client and server. By default, the current directory will be the current directory of the command prompt which runs this command which will be the location, the client and server is ran. This program has limitation though which will be discussed.

Program limitations:

1. This program has a fixed max buffer size for the file content and a fixed size for character array for path. If the clients, file exceeds that of the buffer length of 500, the program will crash with a segmented error as when you try to access memory beyond 500 since the data array size is 500.
2. If the file is empty, the file does not get uploaded or downloaded as the detection is based upon the file length.
3. The max number of clients is 5 concurrently.
4. The client does not have exit command so it can never turn off without the force escape command.

Suggested improvements based upon limitation number:

1. C actually can get dynamic array sizes which could be used for path size and buffer size. In the beginning, we were going to implement a dynamic buffer size, however we came across the problem of memory management namely free and allocating memory. The program originally did not do it properly so there was segment errors. The specification does not have to have dynamic size, we changed it to fixed to simplify the project.
2. There are two ways to improvement directions. First, 1 download/upload the file anyways or 2 prompt the user with a warning. I would suggest doing case 2 as downloading an empty file would be meaningless. To obtain this effect, simply hard code an empty file length case and do either 1 or 2.
3. Change the listen argument to amount of client you want to be able to connect concurrently. The code will be able to handle it as it makes a new child process when it needs to service more than one client. Be careful with the amount of processes, you created as each operating system has a max number of processes that can be created at a time. If you reach this number, the command prompt will not allow you to run create more processes ie clients and you will have to kill some processes. Therefore, the absolute maximum of concurrently running clients would be max number of process /2 as each client will have its own server process. Note, the max number of processes depends on your operating system.

4. Add a new case statement to the for when it detects which command to execute. It should either return 1 so the main function ends or you could call exit for the similar effect.

5.0 – Conclusion

The client and server programs in this laboratory project use the PDU format data to communicate with each other based on the TCP protocol. Each PDU sent has three different fields indicating the type, the length and the data of the data packet to be sent. Since TCP is a connection-oriented protocol, the connection between the client and server must be established for data exchange. The server can receive multiple requests from different clients by putting them into a queue. This project is to examine the connection between one server and one client and learn how the data exchange works between them. Each of the client and server program does its own duties and notify each other during the process. Based on our observation and analysis, the connection and data exchange are implemented successfully and we have examined the processes within the program.

6.0 – Appendix

Client

```
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>

#define SERVER_TCP_PORT 3000
#define BUFLLEN 500
struct PDU {
    char type;
    int length;
    char data[BUFLLEN];
} rpdu, tpdu;
int main(int argc, char *argv[])
{
    char    *host = "localhost"; /* host to use if none supplied */
    char    *bp, sbuf[BUFLLEN], rbuf[BUFLLEN], path[BUFLLEN];
    int     port, bytes_to_read;
    int     sd, n, end = 1; /* socket descriptor and socket type */
    FILE    *fp;
    char cwd[1024];
    struct stat fstat;
    struct hostent *hp; /* pointer to host information entry */
    struct sockaddr_in server; /* an Internet endpoint address */

    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);
    }

    while (end) {
        printf("Please enter a type for input: \nD for download \nU for upload \nP
for path change \nL list path                                \nE error \n");
        scanf(" %c", &tpdu.type);
        tpdu.length = 0; // set default data unit length
        //rpdu data tpdu for signal
        switch(tpdu.type) {
            /* File Download */
            case 'D':
                printf("Enter filename: \n");
                tpdu.length = read(0, tpdu.data, BUFLen-1); // get user
message

```

```

tpdu.data[tpdu.length-1] = '\0';
write(sd, (char *)&tpdu, sizeof(tpdu));
read(sd, (char *)&rpdu, sizeof(rpdu));

if (rpdu.type == 'F') {
    fp = fopen(strcat(tpdu.data,"download"), "w");
    fwrite(rpdu.data, sizeof(char), rpdu.length, fp); //

write data to file

data to write

    if (rpdu.length == BUFLen) { // if there is more

        read(sd, (char *)&rpdu, sizeof(rpdu));
        fwrite(rpdu.data, sizeof(char), rpdu.length,

fp);

    }
    fclose(fp);
    printf("Download Transfer .\n");
} else {
    fprintf(stderr, "%s", rpdu.data);
}
break;

/* File Upload */
case 'U':
    printf("Enter filename:\n");
    tpdu.length = read(0, tpdu.data, BUFLen-1); // get user

message

    tpdu.data[tpdu.length-1] = '\0';
    fp = fopen(tpdu.data, "r"); // open file to be uploaded
    if (fp == NULL) {
        fprintf(stderr, "Error: File \"%s\" not found.\n",

tpdu.data);

    } else {
        write(sd, (char *)&tpdu, sizeof(tpdu)); // make

upload request

        read(sd, (char *)&rpdu, sizeof(rpdu)); // receive

ready signal

        if (rpdu.type == 'R') { // check if server is ready
            tpdu.type = 'F';
            //if(bytes_to_read > 0) { // send data packets
                tpdu.length = fread(tpdu.data,

sizeof(char), BUFLen, fp);

```

```

write(sd, (char *)&tpdu,
sizeof(tpdu));

        //}
        printf("File sent.\n");
    } else {
        fprintf(stderr, "Error: Server not ready.\n");
    }
    fclose(fp);
}
break;

/* Change Directory */
case 'P':
    printf("Enter path: \n");
    n = read(0, tpdu.data, BUFLLEN); // get user message
    tpdu.data[n - 1] = '\0';
    //if(opendir(tpdu.data)) { // check if directory is valid
    //    chdir(tpdu.data);
    //}
    printf("Current working dir: %s\n", getcwd(cwd,
sizeof(cwd)));

    printf("Directory changed to \"%s\"\n", tpdu.data);
    write(sd, (char *)&tpdu, sizeof(tpdu));
    read(sd, (char *)&rpdu, sizeof(rpdu));

    if (rpdu.type = 'R') {
        printf("Directory change .\n");
    } else {
        fprintf(stderr, "Error: Invalid directory.\n");
    }
    break;

/* List Files */
case 'L':
    printf("Enter path: \n");
    n = read(0, tpdu.data, BUFLLEN); // get user message
    tpdu.data[n - 1] = '\0';

    write(sd, (char *)&tpdu, sizeof(tpdu)); // request file list

    read(sd, (char *)&rpdu, sizeof(rpdu)); // recieve file list
    write(1, rpdu.data, rpdu.length); // display to stdout

```

```

        break;

    case 'E':
        rpdu.type = 'Z';
        rpdu.length = 0;
        write(sd, (char *)&rpdu, sizeof(rpdu)); // tell client ready
        read(sd, (char *)&rpdu, sizeof(rpdu));
        if(rpdu.type == 'E')
            printf("Invalid pdu .\n");
        break;
    }
}
close(sd);
return(0);
}

```

Server

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/signal.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>

#define SERVER_TCP_PORT 3000
#define BUFLLEN 500

int echod(int);
void reaper(int);
struct PDU {
    char type;
    int length;
    char data[BUFLLEN];
} rpdu, spdu;

```

```

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()) {
        /* child */
        case 0:
            (void) close(sd);
            exit(echod(new_sd));
        /* parent */
        default:
            (void) close(new_sd);
            break;
        case -1:
            fprintf(stderr, "fork: error\n");
    }
}

/* echod program */
int echod(int sd)
{
    int    bytes_to_read, quit = 1, n, x;
    DIR    *d;
    FILE   *fp;
    char cwd[1024];
    struct dirent *dir;
    struct stat fstat;

    while (quit) {
        read(sd, (char *)&rpdu, sizeof(rpdu)); // data from client
        //rpdu data and spdu is data
        switch(rpdu.type) {
            /* Download request */
            case 'D':
                fp = fopen(rpdu.data, "r");
                if (fp == NULL) {
                    spdu.type = 'E'; // data unit type as error
                    sprintf(spdu.data, "Error: File \"%s\" not found.\n",
rpdu.data);

                    spdu.length = strlen(spdu.data);
                    write(sd, (char *)&spdu, sizeof(spdu));
                    fprintf(stderr, "Error: File \"%s\" not found.\n",
rpdu.data);

```

```

    } else {

        spdu.length = fread(spdu.data, sizeof(char),
BUFLen, fp);

        spdu.type = 'F';
        write(sd, (char *)&spdu, sizeof(spdu));

        printf("Download Transfer of \"%s\" .\n",
rpdu.data);

    }
    break;

/* Upload request */
case 'U':
    spdu.type = 'R';
    spdu.length = 0;

    write(sd, (char *)&spdu, sizeof(spdu)); // tell client ready
    fp = fopen(strcat(rpdu.data,"upload"), "w"); // create file to
write uploaded data

    printf("Starting transfer of \"%s\" .\n", rpdu.data);

    //read data packet
    read(sd, (char *)&rpdu, sizeof(rpdu));
    fwrite(rpdu.data, sizeof(char), rpdu.length, fp); //
write data to file

    printf("Writing upload");

    fclose(fp);
    printf("Upload Transfer .\n");
    break;

/* Change directory */
case 'P':
    if(opendir(rpdu.data)) { // check if
directory is valid

        chdir(rpdu.data);
        printf("Current working dir: %s\n", getcwd(cwd,
sizeof(cwd)));

        printf("Directory changed to \"%s\" .\n", rpdu.data);
        spdu.type = 'R';
    } else { // invalid directory
        spdu.type = 'E';

```

```

        strcpy(spdu.data, "Error: Invalid directory.\n");
        spdu.length = strlen(spdu.data);
    }
    write(sd, (char *)&spdu, sizeof(spdu));    // send reply to
client
        break;

    /* List files */
    case 'L':
        if(opendir(rpdu.data)) {            // check if directory is valid
            d = opendir(rpdu.data);          // open current directory
            spdu.data[0] = '\0';             //strcat need null
            while ((dir = readdir(d)) != NULL) {            // read
contents
                strcat(spdu.data, dir->d_name);
                strcat(spdu.data, "\n");
                printf("%s\n", spdu.data);
            }
            closedir(d); // close directory
            spdu.length = strlen(spdu.data);
            write(sd, (char *)&spdu, sizeof(spdu)); // send list to client
            printf("List files.\n");
        } else { // invalid directory
            spdu.type = 'E';
            strcpy(spdu.data, "Error: Invalid directory.\n");
            spdu.length = strlen(spdu.data);
        }
        break;

    default:
        spdu.type = 'E';
        spdu.length = 0;
        write(sd, (char *)&spdu, sizeof(spdu));
        printf("Invalid pdu.\n");
        break;
    }
}
close(sd); // close TCP connection
return(0);
}

/* reaper */

```

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