Writeup:

Comments:

Code:

CS 4414 Operating Systems – Fall 2017

Homework 5: Simple FTP

Jonathan Colen jc8kf@virginia.edu

The assignment was completed and runs correctly. It can be interacted with via a standard FTP client.

1. Problem Description

The objective of this assignment was to become familiar with remote procedure calls and remote file systems. The program written for this homework implements a simple FTP server with the ability to login, change certain options, list directory entries, and store and retrieve files. The server is implemented according to Section 5.1 the IETF RFC 959 FTP specification with the addition of the LIST command. In addition, only image mode is supported. It can be interacted with through a standard FTP client.

2. Approach

The FTP server implemented in this assignment interacts with the client through a command socket and a data socket and provides basic store, retrieve, and list functionality. Ten commands are supported: USER, QUIT, PORT, TYPE, MODE, STRU, RETR, STORE, NOOP, and LIST. The server program accepts a port number as a command line argument and creates the command socket using the BSD Socket library. The socket is created using the *socket* call, and a *sockaddr* structure is initialized with the port number and any available address. The socket is then bound to the address and port using *bind* and begins to listen for connections using *listen*. After this initialization of the command socket, the program enters the *server_loop* function.

The *server_loop* method keeps track of a single client connection at a time in an indefinite loop. It also stores local variables that keep track of the current mode, structure, transmission type, and user. At each iteration of the loop, the function accepts a new connection on the command socket using *accept*. The connection is verified by sending the client the status code 220 (server ready) and the read loop is entered. At each iteration of this loop, the function

reads a new line into a buffer and executes that line. The program sends status information back to the client through the socket in the form of status codes. The status code format and sequence for each command is written according to the FTP specification. When the read loop ends, the client connection is shut down, the connection is closed, and the server loop iterates.

The read loop performs a different action based on the command, which is contained in the first four characters of the line to be executed. If the command is QUIT, the local variables tracking the mode, structure, type, and user are reset, an exit message is sent to the client, and the read loop is terminated. If the command is NOOP, a confirmation message is sent to the client and the loop iterates without performing the operation. If the command is USER, the program obtains the user name using *sscanf* on the line. The current user variable is then set to this username and a confirmation is sent to the client. No other commands are allowed to be executed until the user has logged in. If any command but these three are called, the program sends status code 530 (not logged in) and the read loop iterates without executing anything else.

The three simplest commands are TYPE, MODE, and STRU, which modify the transmission type, mode, and structure respectively. Each of these commands accepts a character argument, which is stored and checked for validity. Because of the limited implementation of the server, the only allowed type is “I” (image), the only allowed mode is “S” (stream), and the only allowed structure is “F” (file). An attempt to switch to another valid type, mode, or structure results in status code 504 (command not implemented), while an attempt to use an invalid argument results in status code 501 (syntax error in parameters). Otherwise, the local variable tracking the type, mode, or structure is modified and the program sends status code 200 (command okay) to the client.

The PORT command, which specifies the port number for a subsequent data transmission, accepts 6 arguments which specify the address and port number according to the FTP specification. The program sets up a *sockaddr* structure for the address and port specified, sends a confirmation message to the client, and sets a local variable in the read loop indicating that port has been called. The program will not allow calls to STOR, RETR, or LIST, unless this local variable has been set. The *sockaddr* structure is required in order for the data connection to be set up in the execution of these commands.

The data connection setup is done by the *open_data_socket* function. This method takes a *sockaddr* structure as an argument and creates a new socket connected to the designated address

and port number. The file descriptor for this data socket is recorded and stored by the read loop. If the connection fails, the program sends status code 425 (cannot open connection).

The storage and retrieval commands, STOR and RETR, are implemented in a very similar way. They are called using the *ftp_store* and *ftp_retrieve* methods. These methods take the input line from the read loop and the *sockaddr* structure created by the PORT command. Both STOR and RETR accept a file name as an argument. This is obtained from the input line using *sscanf*. The program attempts to open this file name using *open* and stores the resulting file descriptor. If this descriptor is negative, the program sends status code 450 (file could not be created) for STOR and status code 550 (file could not be found) for RETR. Otherwise, the function opens the data socket specified by the *sockaddr* argument using the *open_data_socket* method. For STOR, the function then repeatedly reads from the data connection into a buffer and writes the buffer to the opened file. For RETR, the function repeatedly reads from the opened file into a buffer and writes the buffer to the data connection. The process is repeated until either the data connection finishes sending (for STOR) or the end of file is reached (for RETR). Finally, the methods close the data connection and return to the read loop. The local variable indicating that PORT was called is reset, indicating that a new port must be set for later stores and retrieves.

The final command implemented was LIST, which lists information about a file or all entries in a directory. This command is called using the *ftp_list* method. The *ftp_list* function redirects the */bin/ls* system call to the client through the data connection. This method takes the input line from the read loop and the *sockaddr* structure created by the PORT command. The function scans the input line for a file name argument and stores it in a local filename variable. If none is found, the filename variable is set to “.” for the current working directory. Next, the program opens a data connection using *open_data_socket* and the *sockaddr* argument. A child process is then created using *fork*. The child sets its output to the file descriptor of the data socket and executes “/bin/ls -l <filename>” using *execv*. If this fails, the child process sends the client status code 450 (file action could not be taken) and then terminates. The parent waits for the child process to finish, closes the data connection, resets the port variable as in STOR and RETR, and returns to the read loop.

3. Problems Encountered

The main problem encountered was keeping track of the different connections and ensuring that the port was properly set before storage and retrieval operations. This problem was

eventually fixed by keeping information about the current data connection as a variable in the *server_loop* method so that it could be accessed by later calls by the same client. It was also important to reset the connection information, as well as changes to the transmission type, after each client connection terminated. Forgetting to do these resets resulted in the server sending later information to the wrong port as well as allowing STOR and RETR calls while the client was still in ASCII mode.

4. Testing

The program was tested with the FTP client */usr/bin/ftp*. The server printed out its received commands in order to see how they were ordered and structured. This was also done to aid debugging. The STOR and RETR commands were tested using the *get* and *put* commands in the FTP client. Files were sent to and from the local server and their contents were verified manually. Nonexistent files were also checked for RETR to ensure that the proper status code was sent. The LIST command was checked using *ls*. Use of the *get*, *put*, and *ls* commands indirectly verified the correctness of the PORT implementation. The TYPE command was tested using *type image* in the FTP client, and it was ensured that no storage or retrievals could happen before the type was changed to image mode. USER and QUIT were checked on startup and termination of the FTP client, as it would not start up or terminate without proper implementation of these two commands. Several sessions were opened sequentially to ensure that information such as the transmission type, username, and data connection information were reset upon quitting.

The MODE, STRU, and NOOP commands were checked using the *nc* tool, as the standard FTP client did not allow transmission of these commands. The response codes for these commands were checked to ensure they were correct for a variety of arguments.

5. Conclusions

The assignment was successful in teaching remote procedure calls and remote file system management. I have learned how to use the Berkeley Sockets API and the FTP specification to support basic file transfer commands with a remote client. The FTP server works as expected and interacts correctly with the */usr/bin/ftp* client.

6. Pledge

On my honor as a student I have neither given nor received aid on this assignment.

```
/**
*       Name:            Jonathan Colen
*       Email:           jc8kf@virginia.edu
*       Class:           CS 4414
*       Professor:       Andrew Grimshaw
*   Assignment: Machine Problem 5
*
*   The purpose of this assignment is to implement a bare bones ftp server.
*   This server supports storage and retrieval of files as well as directory
*   listings.
*
*   This program can be compiled via make and run via
        ./my_ftpd <port number>
*   The server can be connected to with the command
        /usr/bin/ftp localhost <port number>
*/


#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

int buff_size = 256; //Maximum line buffer size (input and output)

/**
* Open a data connection socket and connect it to a desired address
* @param socketfd Where the socket file descriptor will be stored
* @param sa Information about the socket to be connected to
* @param commandfd File descriptor for the command connection
*/
int open_data_socket(int * socketfd, struct sockaddr_in sa, int commandfd) {
    char outbuffer[buff_size];  //Stores command connection output
    strcpy(outbuffer, "150 File status okay; about to open data connection\r\n");
    send(commandfd, outbuffer, strlen(outbuffer), 0);
    *socketfd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);  //Initialize socket
    if (*socketfd == -1)    {
        //Handle socket failure
        strcpy(outbuffer, "425 Can't open data connection\r\n");
        send(commandfd, outbuffer, strlen(outbuffer), 0);
        return -1;
    }

    //Connect socket to address
    if (connect(*socketfd, (struct sockaddr *)&sa, sizeof sa) == -1)    {
        //Handle connect failure
        strcpy(outbuffer, "425 Can't open data connection\r\n");
        send(commandfd, outbuffer, strlen(outbuffer), 0);
        return -1;
    }

    return 1;
}

/**
* Execute the TYPE command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
*/
void change_type(int connectfd, char * buffer, char * type) {
    char newtype, filler;   //TYPE techically has two character arguments
    int match = sscanf(buffer, "TYPE %c %c", &newtype, &filler);
    char outbuffer[buff_size];
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");
    printf("TYPE %c %c\n", newtype, filler);
    if (match > 0)  {
        if (newtype == 'I') {   //Only allow image mode
            //Switch type
            *type = newtype;
            strcpy(outbuffer, "200 Command okay\r\n");
        } else if (newtype == 'A' || newtype == 'E')   {
            //Different return code for unsupported commands
            //At least check to see second character is valid before rejecting the command
            if (match > 1 && (filler == 'N' || filler == 'T' || filler == 'C'))    {
                strcpy(outbuffer, "504 Command not implemented for that parameter\r\n");
            }
        } else if (newtype == 'L') {
            strcpy(outbuffer, "504 Command not implemented for that parameter\r\n");
        }
    }

    //Send output to client
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}
```

```
/**
* Execute the MODE command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param mode Storage point for new mode
*/
void change_mode(int connectfd, char * buffer, char * mode) {
    char newmode;   //Stores new mode
    int match = sscanf(buffer, "MODE %c", &newmode);
    char outbuffer[buff_size];
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");   //Default response
    printf("MODE %c\n", newmode);
    if (match > 0)  {
        if (newmode == 'S') {   //Only allow stream mode
            *mode = newmode;
            strcpy(outbuffer, "200 Command okay\r\n");
        } else if (newmode == 'B' || newmode == 'C')    {
            strcpy(outbuffer, "504 Command not implemented for that parameter\r\n");
        }
    }

    //Send output to client
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Execute the STRU command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param stru Storage point for new structure type
*/
void change_stru(int connectfd, char * buffer, char * stru) {
    char newstru;   //Stores new structure
    int match = sscanf(buffer, "STRU %c", &newstru);
    char outbuffer[buff_size];  //stores output to client
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");   //Default response
    printf("STRU %c\n", newstru);
    if (match > 0)  {
        if (newstru == 'F') {   //Only allow file structure
            *stru = newstru;
            strcpy(outbuffer, "200 Command okay\r\n");
        } else if (newstru == 'R' || newstru == 'P')    {
            strcpy(outbuffer, "504 Command not implemented for that parameter\r\n");
        }
    }

    //Send output to client
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Execute the PORT command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param data_sa The socket address information to be filled in
* @param port A field saying whether port has been called, to be set to 1 on success
*/
void setup_port(int connectfd, char * buffer, struct sockaddr_in * data_sa, int * port)  {
    int a1, a2, a3, a4, p1, p2; //Address and port numbers
    int match = sscanf(buffer, "PORT %d,%d,%d,%d,%d,%d",
        &a1, &a2, &a3, &a4, &p1, &p2);  //Scan for connection information
    char outbuffer[buff_size];  //stores output to client
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");   //Default response
    if (match == 6) {
        printf("PORT %d,%d,%d,%d,%d,%d\n",
            a1, a2, a3, a4, p1, p2);
        //Initialize data in sockaddr structure
        memset(data_sa, 0, sizeof(*data_sa));
        data_sa->sin_family = AF_INET;
        data_sa->sin_port = htons(p1 * 256 + p2);   //From the Berkeley sockets wikipedia page
        char addr[100];
        sprintf(addr, "%d.%d.%d.%d", a1, a2, a3, a4);
        int res = inet_pton(AF_INET, addr, &(data_sa->sin_addr));   //Also from Berkeley sockets wikipedia
        *port = 1;  //Set that port has been called successfully
        strcpy(outbuffer, "200 Command okay\r\n");
    }

    //Send output to client
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Execute the RETR command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param data_sa Information about the data connection
* @param port Whether port has been called prior to this
*/
void ftp_retrieve(int connectfd, char * buffer, struct sockaddr_in data_sa, int * port)    {
    int data_socket;    //file descriptor for data connection socket
    char filename[buff_size];   //stores filename
    char databuffer[buff_size]; //stores data to be written to client
```

```
    char outbuffer[buff_size];  //stores output

    memset(filename, 0, buff_size);
    int match = sscanf(buffer, "RETR %s\n", filename);  //Scan for filename
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");   //Default response
    if (match == 1) {
        printf("RETR %s\n", filename);
        int fd = open(filename, O_RDONLY);
        if (fd < 0) {   //If file does not exist, exit
            strcpy(outbuffer, "550 Requested action not taken. File unavailable\r\n");
        } else  {
            int ret = open_data_socket(&data_socket, data_sa, connectfd);   //Initialize data connection
            if (ret == 1)   {
                int bytes;
                bytes = read(fd, databuffer, buff_size);    //Write to socket until end of file
                while (bytes > 0)   {
                    send(data_socket, databuffer, bytes, 0);
                    bytes = read(fd, databuffer, buff_size);
                }
                close(fd);
                //Close connection
                strcpy(outbuffer, "226 Closing data connection. Requested file action successful\r\n");
                shutdown(data_socket, SHUT_RDWR);
                close(data_socket);
            }
        }
    }

    //Reset port and send exit code to client
    *port = 0;
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Execute the STOR command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param data_sa Information about the data connection
* @param port Whether port has been called prior to this
*/
void ftp_store(int connectfd, char * buffer, struct sockaddr_in data_sa, int * port)  {
    int data_socket;    //file descriptor for data connection socket
    char filename[buff_size];   //stores filename
    char databuffer[buff_size]; //stores data to be written to server
    char outbuffer[buff_size];  //stores output

    memset(filename, 0, buff_size);
    int match = sscanf(buffer, "STOR %s", filename);    //Scan for filename
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");   //Default response
    if (match == 1) {
        printf("STOR %s\n", filename);
        int fd = open(filename, O_WRONLY | O_CREAT);
        if (fd < 0) {   //If file cannot be created exit
            strcpy(outbuffer, "450 Requested action not taken. File could not be created\r\n");
        } else  {
            int ret = open_data_socket(&data_socket, data_sa, connectfd);   //Initialize data connection
            if (ret == 1)   {
                int bytes;
                bytes = read(data_socket, databuffer, buff_size);   //Write to new file until no more data comes
                while(bytes > 0)    {
                    write(fd, databuffer, bytes);
                    bytes = read(data_socket, databuffer, buff_size);
                }
                close(fd);
                //Close connection
                strcpy(outbuffer, "226 Closing data connection. Requested file action successful\r\n");
                shutdown(data_socket, SHUT_RDWR);
                close(data_socket);
            }
        }
    }

    //Reset port and send exit code to client
    *port = 0;
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Execute the LIST [<filename>] command
* @param connectfd The command port file descriptor
* @param buffer The line to be executed
* @param data_sa Information about the data connection
* @param port Whether port has been called prior to this
*/
void ftp_list(int connectfd, char * buffer, struct sockaddr_in data_sa, int * port)  {
    int data_socket;    //file descriptor for data connection socket
    char filename[buff_size];   //stores filename
    char outbuffer[buff_size];  //output buffer

    memset(filename, 0, buff_size); //avoid memory overlaps

    int match = sscanf(buffer, "LIST %s", filename);    //Scan for filename
```

```
    //Default response. Will be changed on success
    strcpy(outbuffer, "501 Syntax error in parameters or arguments\r\n");

    //Default to cwd
    if (match == 0 || strlen(filename) == 0) {
        strcpy(filename, ".");
    }

    printf("LIST %s\n", filename);
    int ret = open_data_socket(&data_socket, data_sa, connectfd);   //Open data connection
    if (ret == 1)   {
        //Fork a child process and output ls -l <filename> to the data socket
        int pid = fork();

        //If child process, execute ls -l <filename> and send to data socket
        if (pid == 0)   {
            dup2(data_socket, 1);   //dup2 to data connection
            char * args[4];         //Set up args for /bin/ls
            args[0] = malloc(sizeof(char) * 10);
            args[1] = malloc(sizeof(char) * 10);
            args[2] = malloc(sizeof(char) * buff_size);
            args[3] = NULL;         //NULL terminate
            strcpy(args[0], "/bin/ls");
            strcpy(args[1], "-l");
            strcpy(args[2], filename);
            int err = execv(args[0], args);
            if (err == -1)  {
                //Handle error
                printf("ls -l returned exit code %d\n", err);
                strcpy(outbuffer, "450 Requested file action not taken\r\n");
                send(connectfd, outbuffer, strlen(outbuffer), 0);
                exit(1);
            }
        }

        waitpid(pid, NULL, 0);  //Wait for child

        //Close data connection
        strcpy(outbuffer, "226 Closing data connection. Requested file action successful\r\n");
        shutdown(data_socket, SHUT_RDWR);
        close(data_socket);
    }

    //Reset port and send output code to client
    *port = 0;
    send(connectfd, outbuffer, strlen(outbuffer), 0);
}

/**
* Main server loop. Accepts a socket file descriptor as argument.
* Handles the interaction with the clients in an indefinite loop
*/
void server_loop(int socketfd)  {
    //Keep track of certain variables
    char mode = 'S';    //Current mode (starts as stream)
    char type = 'A';    //Current type (starts as ASCII)
    char stru = 'F';    //Current structure (starts as file)
    char username[buff_size];   //Stores current user
    int login = 0;      //Tracks whether someone is logged in

    char buffer[buff_size];     //Input buffer
    char outbuffer[buff_size];  //Output buffer

    int port = 0;               //Tracks whether port has been called
    struct sockaddr_in data_sa; //Tracks data socket information

    //Connection loop
    for (;;)    {
        //Accept a new connection
        int connectfd = accept(socketfd, NULL, NULL);

        //Check connection was successful
        if (0 > connectfd)  {
            perror("accept failed");
            close(socketfd);
            exit(EXIT_FAILURE);
        }

        //Tell connection server is ready
        send(connectfd, "220 my FTP server\r\n", 19, 0);

        //Read loop
        while(1)    {
            memset(buffer, 0, buff_size);
            read(connectfd, buffer, buff_size); //Read command into buffer
            //End read loop if QUIT and move to socket termination
            if (strncmp(buffer, "QUIT", 4) == 0)    {
                //Reset all variables
                login = 0;
                port = 0;
                mode = 'S';
                type = 'A';
```

```
                stru = 'F';
                strcpy(outbuffer, "221 Service closing control connection\r\n");
                send(connectfd, outbuffer, strlen(outbuffer), 0);
                printf("QUIT\n");
                break;
            }
            //Do nothing if NOOP
            if (strncmp(buffer, "NOOP", 4) == 0)    {
                send(connectfd, "200 Command okay\r\n", 18, 0);
                printf("NOOP\n");
                continue;
            }
            //Handle user login
            if (strncmp(buffer, "USER", 4) == 0)    {
                int match = sscanf(buffer, "USER %s", username);
                printf("USER %s\n", username);
                if (match > 0)  {
                    login = 1;
                    sprintf(outbuffer, "230 User logged in as %s, proceed\r\n", username);
                    send(connectfd, outbuffer, strlen(outbuffer), 0);
                    continue;
                }
                strcpy(outbuffer, "500 Syntax error, command unrecognized\r\n");
                send(connectfd, outbuffer, strlen(outbuffer), 0);
            }
            //Don't allow any other commands until user is authenticated
            if (!login) {
                strcpy(outbuffer, "530 Not logged in\r\n");
                send(connectfd, outbuffer, strlen(outbuffer), 0);
                continue;
            }
            //All other commands are handled by individual methods
            if (strncmp(buffer, "TYPE", 4) == 0)    {
                change_type(connectfd, buffer, &type);
            } else if (strncmp(buffer, "MODE", 4) == 0)    {
                change_mode(connectfd, buffer, &mode);
            } else if (strncmp(buffer, "STRU", 4) == 0)    {
                change_stru(connectfd, buffer, &stru);
            } else if (strncmp(buffer, "PORT", 4) == 0)    {
                setup_port(connectfd, buffer, &data_sa, &port);
            } else if (strncmp(buffer, "RETR", 4) == 0) {
                if (type != 'I')    {   //Abort if not in image mode
                    strcpy(outbuffer, "451 Requested action aborted: local error in processing\r\n");
                    send(connectfd, outbuffer, strlen(outbuffer), 0);
                } else if (port)    {
                    ftp_retrieve(connectfd, buffer, data_sa, &port);
                }
            } else if (strncmp(buffer, "STOR", 4) == 0) {
                if (type != 'I')    {   //Abort if not in image mode
                    strcpy(outbuffer, "451 Requested action aborted: local error in processing\r\n");
                    send(connectfd, outbuffer, strlen(outbuffer), 0);
                    continue;
                } else if (port)    {
                    ftp_store(connectfd, buffer, data_sa, &port);
                }
            } else if (strncmp(buffer, "LIST", 4) == 0) {
                if (port)   {
                    ftp_list(connectfd, buffer, data_sa, &port);
                }
            }
            else {
                //Return code 500 if command is not recognized
                strcpy(outbuffer, "500 Syntax error, command unrecognized\r\n");
                send(connectfd, outbuffer, strlen(outbuffer), 0);
            }
        }
        //Shutdown connection with client
        if (shutdown(connectfd, SHUT_RDWR) == -1)   {
            perror("shutdown failed");
            close(socketfd);
            close(connectfd);
            exit(EXIT_FAILURE);
        }

        close(connectfd);
    }

}

/**
* Main method. Accept as argument a port number and open a socket on that port number
* This method also calls the main server loop
*/
int main(int argc, char ** argv)  {
    //Get port number from arguments
    if (argc < 2)   {
        printf("Please supply port number\n");
        exit(1);
    }

    int main_port = atoi(argv[1]);
```

```
    //Create socket
    struct sockaddr_in sa;
    int SocketFD = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (SocketFD == -1) {
        perror("cannot create socket");
        exit(EXIT_FAILURE);
    }

    //Initialize socket address and port
    memset(&sa, 0, sizeof sa);

    sa.sin_family = AF_INET;
    sa.sin_port = htons(main_port);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);

    //Bind socket to address/port
    if (bind(SocketFD, (struct sockaddr *)&sa, sizeof sa) == -1)    {
        perror("bind failed");
        close(SocketFD);
        exit(EXIT_FAILURE);
    }

    //Start listening on socket
    if (listen(SocketFD, 10) == -1) {
        perror("listen failed");
        close(SocketFD);
        exit(EXIT_FAILURE);
    }

    //Call main server loop
    server_loop(SocketFD);
    //Close socket on exit
    close(SocketFD);

    return EXIT_SUCCESS;
}
```