A Mini Project Report
On

FILE TRANSFER USING CLIENT AND SERVER MODEL

By

SYED MAHBOOB ALI **1602-18-733-055**

TASSAIN RASOOL MALLIK **1602-18-733-035**

Department of Computer Science & Engineering

Vasavi College of Engineering (Autonomous)
(Affiliated to Osmania University)
Ibrahimbagh, Hyderabad-31

2020-21

ACKNOWLEDGEMENT

We would like to express our heartfelt gratitude to T. Naishitha Reddy, our project guide, for her valuable guidance and constant support, along with her capable instructions and persistent encouragement.

We are grateful to our Head of Department, Dr. T. Adilakshmi, for her steady support and the provision of every resource required for the completion of this project.

We would like to take this opportunity to thank our Principal, Dr. S. V. Ramana, as well as the management of the institute, for having designed an excellent learning atmosphere.

We are thankful to and fortunate enough to get constant encouragement, support and guidance which helped us in successfully completing our project work.

ABSTRACT

File transfer is the process of copying or moving a file from a computer to another over a network or Internet connection. The basic idea is to create a server that listens on a particular port, this server will be responsible for receiving files (you can make the server sends files as well). On the other hand, the client will try to connect to the server and send a file of any type.

Here we can send multiple files at a time and share any kind of file at any given time. The file that we are sending gets divided into various parts and these parts gets transmitted one by one. Each part size is predefined and based on that the number of parts are decided.

Data Security has become a necessity in a LAN based system. It is very easy for an intruder to interrupt file transfer in a closed LAN system. There are number of clients connected to a server in a closed network. The clients can easily intrude the files being sent between a client and the server. In this project we will be using FTP to send data to the server. The project is divided into modules. Clients can send shared data using File Transfer Protocol. Similarly clients can also send compressed data to the server. If interrupted by any intruder, the compressed file will only contain a garbage value i.e. the original data is secured. For communication purpose, we will be incorporating a client

TABLE OF CONTENTS

LIST OF FIGURES

TOPIC **PAGE NO**

INTRODUCTION

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

Stages for server

- **Socket creation:**

  int sockfd = socket(domain, type, protocol)

  sockfd: socket descriptor, an integer (like a file-handle)
  domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)
  type: communication type
  SOCK_STREAM: TCP(reliable, connection oriented)
  SOCK_DGRAM: UDP(unreliable, connectionless)
  protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)
- **Setsockopt:**
- int setsockopt(int sockfd, int level, int optname,

  const void *optval, socklen_t optlen);

  This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

- **Bind:**
- int bind(int sockfd, const struct sockaddr *addr,

  socklen_t addrlen);

  After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

- **Listen:**

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

- **Accept:**
  int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

  It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Stages for Client

- **Socket connection:** Exactly same as that of server's socket creation
- **Connect:**
- int connect(int sockfd, const struct sockaddr *addr,

  socklen_t addrlen);

  The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

In our project we can send multiple files at a time and share any kind of file at any given time. The file that we are sending gets divided into various parts and these parts gets transmitted one by one. Each part size is predefined and based on that the number of parts are decided.Our project explicitly uses sockets and ports. You can select the port number you want. This gives us much greater control over the computer network and how we transfer dataIt transfers files in parts and has much greater extensibility. For example, you could use this to transfer files programmatically over the internet and instead of us using the default ports of 80 or 443 for HTTP or HTTPS, we get to choose which port to transfer from. This also allows us to pause the data transfer. For example, if 3/8 parts are done, then we can pause there and upon resuming, the file transfer will again start from the 4/8th part.

IMPLEMENTATION OF SOURCE CODE

SERVER.C

```
#include <stdio.h>

#include <sys/socket.h>

#include <sys/types.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <netinet/in.h>

#include <netdb.h>

#include <fcntl.h>

#include <signal.h>

#define BUF_SIZE 200

#define STEP_SIZE 20

#define FINISH_CODE "432"

#define ACK_CODE "312"


#define PORTNO 51712


int sockfd, newsockfd;


int thr_error (const char *msg) {

        perror(msg);
```

```
        exit(1);
}

void shut_down (int sig_num) {
        printf("Shutting down server\n");
        close(newsockfd);
        close(sockfd);
        exit(1);
}

int send_file (int newsockfd, char *filename) {
        int fd = open(filename, O_RDONLY);
        if (fd < 0) {
                write(newsockfd, "-1", strlen("-1"));
                return -1;
        }
        printf("Sending file '%s'\n", filename);
        int f_size = lseek(fd, 0, SEEK_END);
        int offset = 0, n_steps = f_size/STEP_SIZE, to_read = STEP_SIZE;
        char buffer[STEP_SIZE+1];
        if (f_size%STEP_SIZE != 0) n_steps++;
        // printf("Filename %s, file size = %d, n_steps = %d\n", filename, f_size, n_steps);
```

```
// Send no. of reads it will take to get the entire file to client
printf("N_steps = %d\n", n_steps);
char msg[100];
sprintf(msg, "%d", n_steps);
int w_bytes = write(newsockfd, msg, strlen(msg));
if (w_bytes < 0) { perror("Error writing no. of steps for file to socket"); }
bzero(buffer, STEP_SIZE);
int a_bytes = read(newsockfd, buffer, STEP_SIZE);
if (strcmp(buffer, ACK_CODE) != 0) perror("Did not receive acknowledgement from client");
fflush(stdout);

// Read and send file in windows with incremental offset
int cc = 0;
for (offset = 0; offset < f_size; offset += STEP_SIZE) {
        if (offset+STEP_SIZE > f_size) {
                to_read = f_size - offset;
        }
        bzero(buffer, STEP_SIZE);
        lseek(fd, offset, SEEK_SET);
        read(fd, buffer, to_read);
        w_bytes = write(newsockfd, buffer, strlen(buffer));
        if (w_bytes < 0) { perror("Error writing file to socket"); }
```

```
                // printf("\nData %d:\n%s\n", cc, buffer);


                // Receive acknowledgement
                bzero(buffer, STEP_SIZE);
                a_bytes = read(newsockfd, buffer, STEP_SIZE);
                if (strcmp(buffer, ACK_CODE) != 0) perror("Did not receive acknowledgement from client");
                fflush(stdout);


                cc++;
        }
        w_bytes = write(newsockfd, FINISH_CODE, strlen(buffer));
        // printf("Ran %d times\n", cc);
        printf("Finished sending file '%s'\n", filename);
        return 1;
}


int main (int *argc, char *argv[]) {
        // int sockfd;
        signal(SIGINT, shut_down);
        sockfd = socket(AF_INET, SOCK_STREAM, 0);


        struct sockaddr_in serv_addr;
```

```
int portno = PORTNO;


bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);


int bind_ret = bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
if (bind_ret < 0) { thr_error("Bind error on server"); }
listen(sockfd, 10);


int active = 1;
while ( active == 1 ) {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &client_addr, &client_len);
        if (newsockfd < 0) { thr_error("Error while accepting connection"); }


        printf("Connection accepted from client\n");
        char *buffer = (char *) malloc (sizeof(char) * BUF_SIZE);
        int r_bytes, w_bytes;
        // printf("Waiting for write to file\n");
```

```
            int fl = 0;
            while (fl == 0) {
                // printf("\tFile loop running\n");
                r_bytes = read(newsockfd, buffer, (size_t) BUF_SIZE);
                if (r_bytes < 0) { perror("Erorr reading from file"); fl = 1; }
                else if (r_bytes == 0) { printf("Client disconnected\n"); fflush(stdout); fl =
1; }
                // printf("Received message %s\n", buffer);
                fflush(stdout);
                if (strcmp(buffer, "__exit__") == 0) { fl = 1; }

                send_file(newsockfd, buffer);
                /*
                char msg[100];
                sprintf(msg, "Acknowledged file '%s'\n", buffer);
                w_bytes = write(newsockfd, msg, strlen(msg));
                if (w_bytes < 0) { perror("Error writing filename to socket"); }
                */

                bzero(buffer, BUF_SIZE);
            }
            close(newsockfd);
        }
```

```
        close(sockfd);


        printf("Connection closed\n");
        return 0;
}
```

CLIENT.C

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <netdb.h>
#include <fcntl.h>
#define BUF_SIZE 300
#define CMD_SIZE 200
#define FN_SIZE 300
#define FINISH_CODE "432"
#define ACK_CODE "312"
```

```
#define PORTNO 51712



int sockfd;

int thr_error (const char *msg) {
        perror(msg);
        exit(1);
}


int parse_buffer (char *buffer, char *files[], int *n_files) {
        char *delim = " ", *tokens[100], *tok, *cmd = (char *) malloc (sizeof(char) *
CMD_SIZE);
        tok = strtok(buffer, delim);
        strcpy(cmd, tok);
        // printf("Received cmd: %s\n", cmd);
        if (strcmp(cmd, "get") != 0) {
                printf("Invalid command. Please try again\n");
                return -1;
        }


        tok = strtok(NULL, delim);
        int i = 0;
```

```
	while (tok != NULL) {
		files[i] = (char *) malloc (sizeof(char) * FN_SIZE);
		strcpy(files[i], tok);
		tok = strtok(NULL, delim);
		// printf("File %d: %s\n", i, files[i]);
		i++;
	}

	*n_files = i;

	return 1;
}

void write_to_file (char *filename, char *buf, int n_bytes) {
	int wfd = open(filename, O_RDWR | O_CREAT | O_APPEND, 0600);
	if (wfd < 0) { thr_error("Unable to open output file"); exit(1); }
	write(wfd, buf, n_bytes);
	close(wfd);
}

int receive_file (int sockfd, char *filename) {
	char *buffer = (char *) malloc (sizeof(char) * BUF_SIZE);
	char *outfile = (char *) malloc (sizeof(char) * BUF_SIZE);
```

```
int start_i = 0;
for (int i = 0; i < strlen(filename); i++) {
        if (filename[i] == '/') start_i = i+1;
}
sprintf(outfile, "%s", &filename[start_i]);

remove(outfile);

// Reading total number of reads it will take
int w_bytes;
int r_bytes = read(sockfd, buffer, (size_t) BUF_SIZE);
if (r_bytes < 0) { thr_error("Error getting data from socket"); }
int n_reads = atoi(buffer);
if (n_reads == -1) {
        printf("File '%s' does not exist\n", filename);
        return -1;
}
w_bytes = write(sockfd, ACK_CODE, strlen(ACK_CODE));
if (w_bytes < 0) { thr_error("Error sending acknowledgement to socket"); }

float progress;
int bars;
```

```
// Reading for n_reads times sent by server and
// writing to outfile simultaneously
for (int i = 0; i < n_reads; i++) {
        r_bytes = read(sockfd, buffer, (size_t) 40);
        if (r_bytes < 0) { thr_error("Error getting data from socket"); }

        // Send acknowledgement for received bytes
        w_bytes = write(sockfd, ACK_CODE, strlen(ACK_CODE));
        if (w_bytes < 0) { thr_error("Error sending acknowledgement to socket"); }

        write_to_file(outfile, buffer, r_bytes);
        fflush(stdout);

        progress = (((float)(i+1)) / (float) n_reads) * 100;
        bars = (int) progress/4;
        printf("Progress:  %3.3f%%\t", progress);

        printf("[");
        for (int i = 0; i < 25; i++) {
                if (i < bars) printf("=");
                else printf(" ");
        }
```

```
            printf("]");
            printf("\r");
        }
        printf("Progress: 100.000%% \t");
        printf("[");
        for (int i = 0; i < 25; i++) {
            printf("=");
        }
        printf("]\n");

        bzero(buffer, (size_t) BUF_SIZE * sizeof(char));
        r_bytes = read(sockfd, buffer, (size_t) BUF_SIZE);
        // printf("Received finished msg: %s\n", buffer);
        if ((r_bytes < 0) || (strcmp(buffer, FINISH_CODE) != 0)) {
            printf("Received finished code %s\n", buffer);
            thr_error("Error in getting finished acknowledgement");
        }

        printf("Finished writing to file %s\n", outfile);
        return 1;
}

int request_files(char *files[], int n_files) {
```

```
        int r_bytes, w_bytes;
        char *buffer = (char *) malloc (sizeof(char) * BUF_SIZE);
        for (int i = 0; i < n_files; i++) {
                printf("Requesting file '%s'\n", files[i]);
                w_bytes = write(sockfd, files[i], strlen(files[i]));
                if (w_bytes < 0) { thr_error("Error writing filename to socket"); }

                bzero(buffer, BUF_SIZE);
                receive_file(sockfd, files[i]);
        }
        return 1;
}

int main (int argc, char *argv[]) {
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd == -1) { thr_error("Unable to create socket from client side"); }

        struct sockaddr_in server_addr;
        struct hostent *server;
        int portno = PORTNO;

        server = gethostbyname("localhost");
        if (server == NULL) { thr_error("localhost not found by client"); }
```

```
bzero((char *) &server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
bcopy((char *) server->h_addr, (char *) &server_addr.sin_addr.s_addr, server->h_length);
server_addr.sin_port = htons(portno);

int connection = connect(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
if (connection < 0) { thr_error("Unable to establish connection with server. Please check port number defined in the files"); }

printf("Successfully established connection\n");

int fl = 0;
size_t buf_size = BUF_SIZE;
char *buffer = (char *) malloc (sizeof(char) * BUF_SIZE);
char *files[100];
int n_files = 0;
while (fl == 0) {
        printf("client> ");
        int chars = getline(&buffer, &buf_size, stdin);
        if (chars != 0) {
                // printf("Received in buffer: %s\n", buffer);
                if (strcmp(buffer, "\n") == 0) continue;
```

```
                buffer[chars-1] = '\0';
                if (strcmp(buffer, "exit") == 0) { fl = 1; }
                else {
                        if (parse_buffer(buffer, files, &n_files) != -1)
                                request_files(files, n_files);
                }
        }
    }

    printf("Exiting\n");

    close(sockfd);

    return 0;
}
```

OUTPUT / SCREENSHOTS

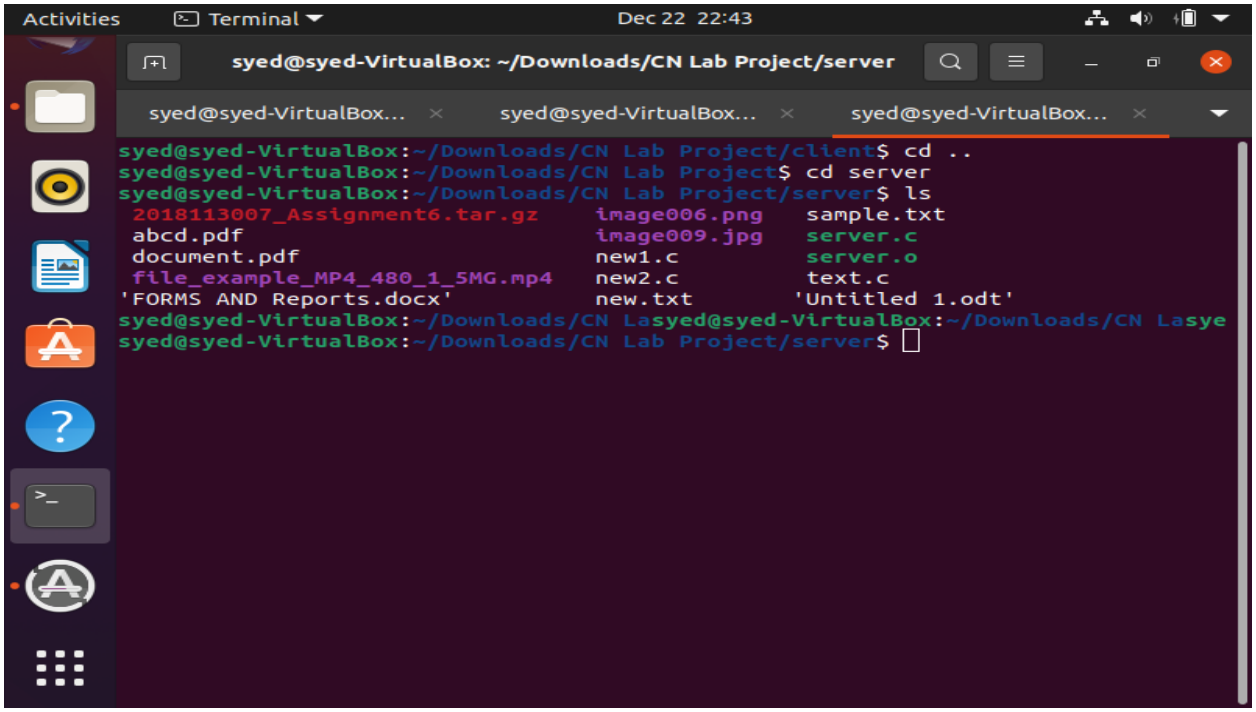

Figure 1: List of files in server side

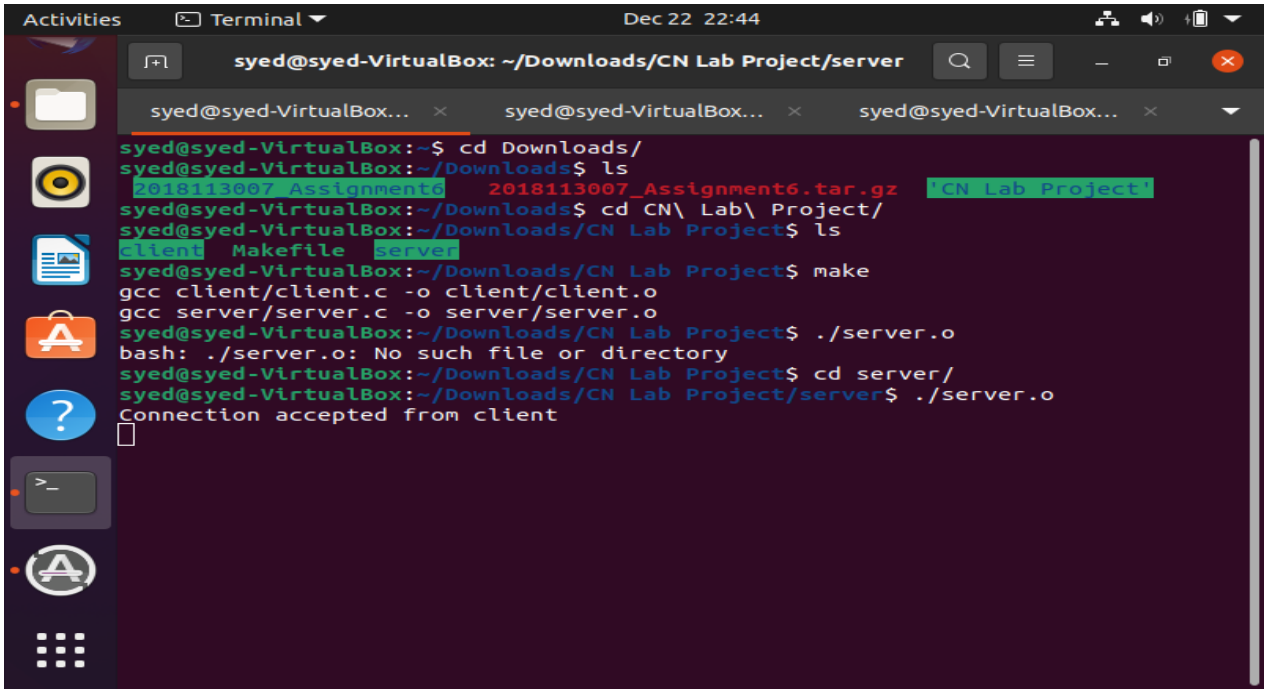

Figure 2: Server side connection

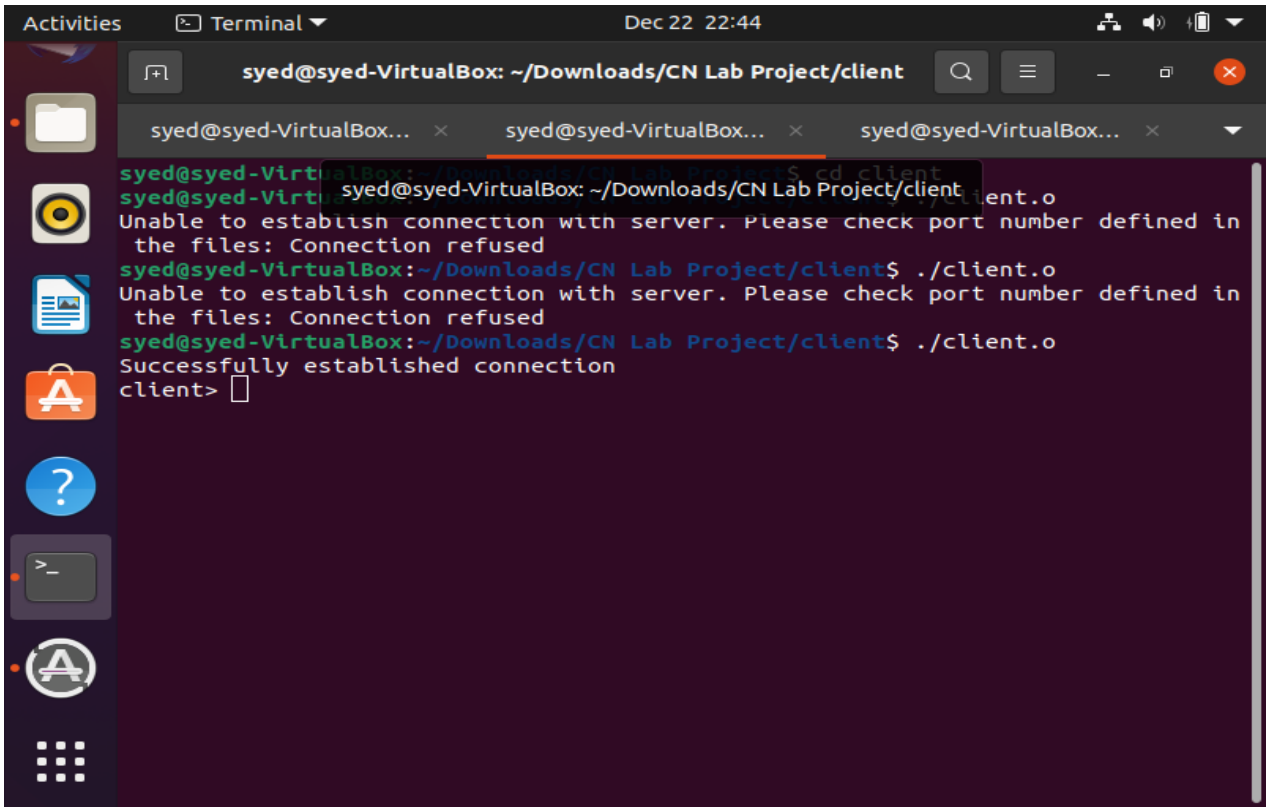

Figure 3: Client side connection

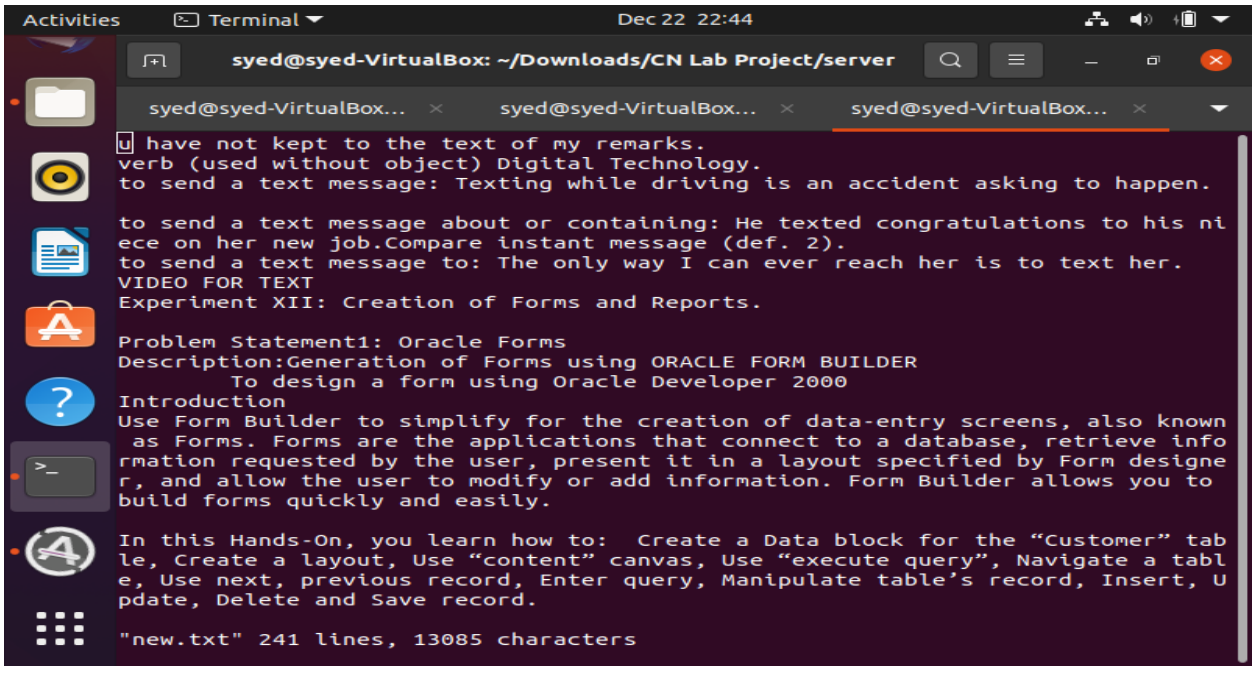

Figure 4: Example Text file

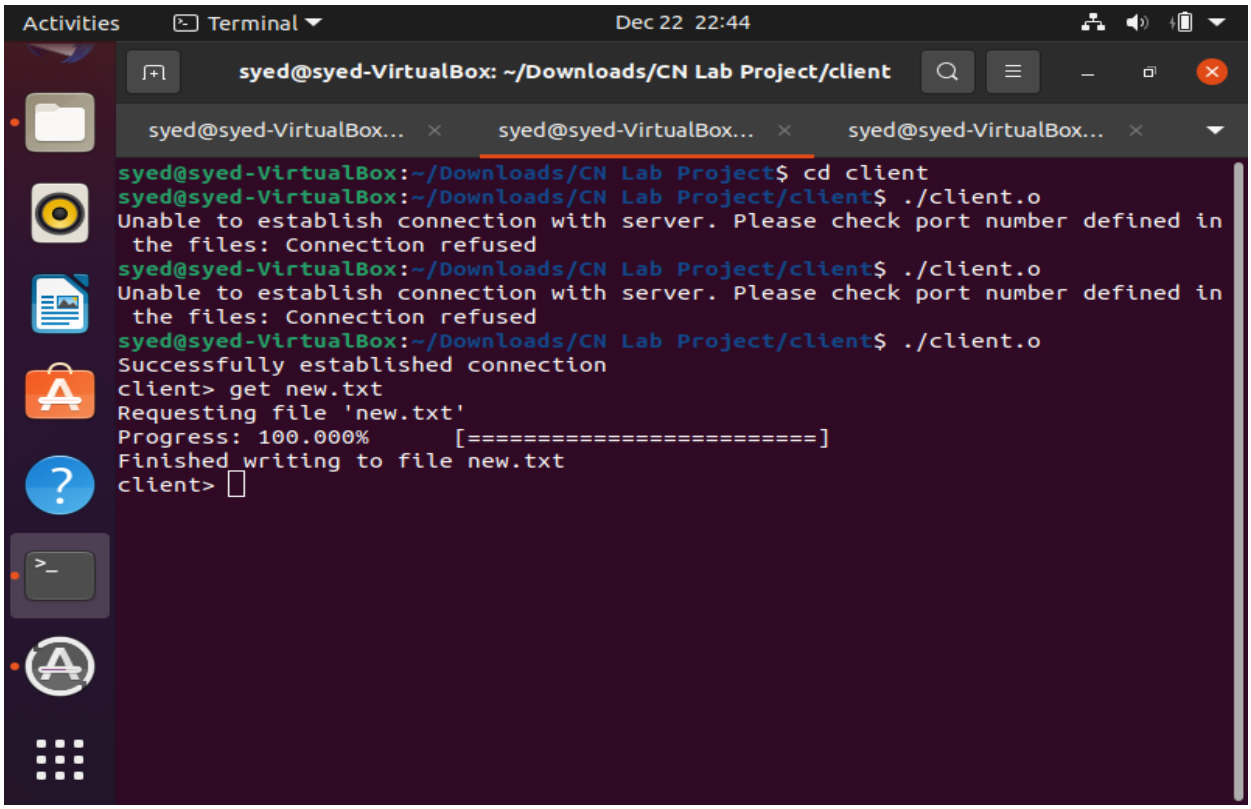

Figure 5 : Get file at client side

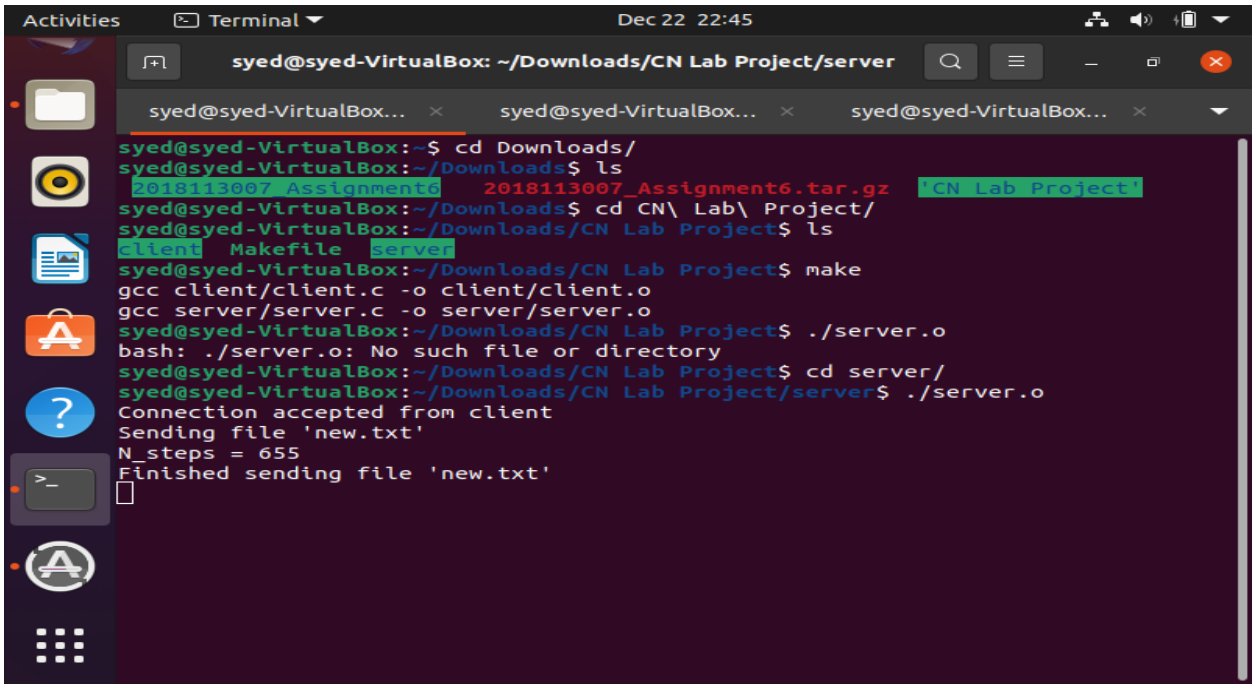

Figure 6 : Finished sending one file

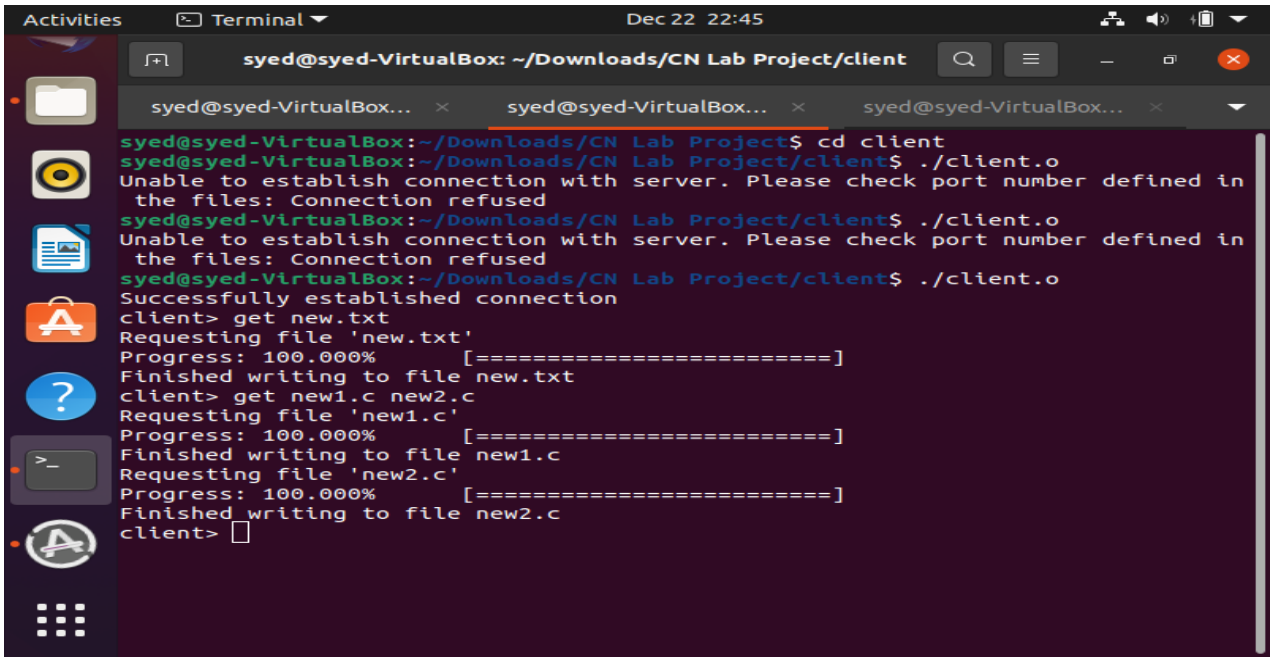

Figure 7 : Get multiple files at client side

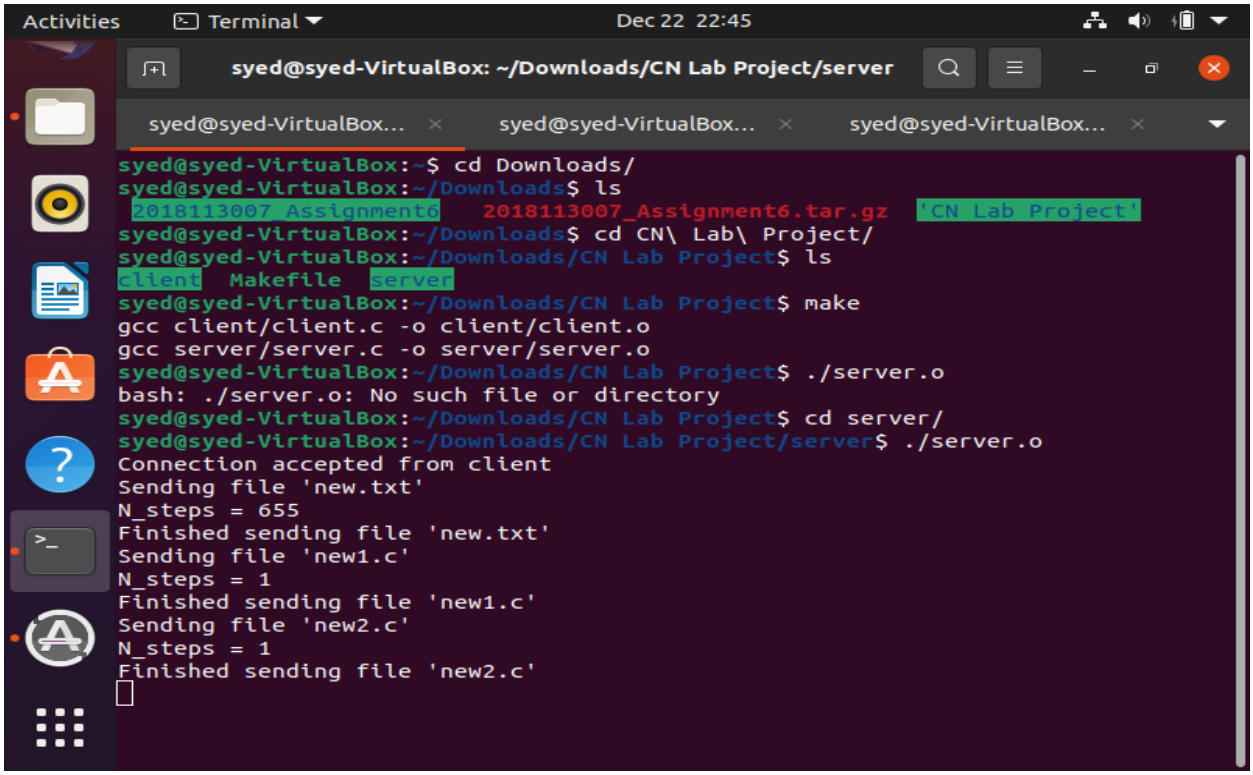

Figure 8 : Finished sending multiple files

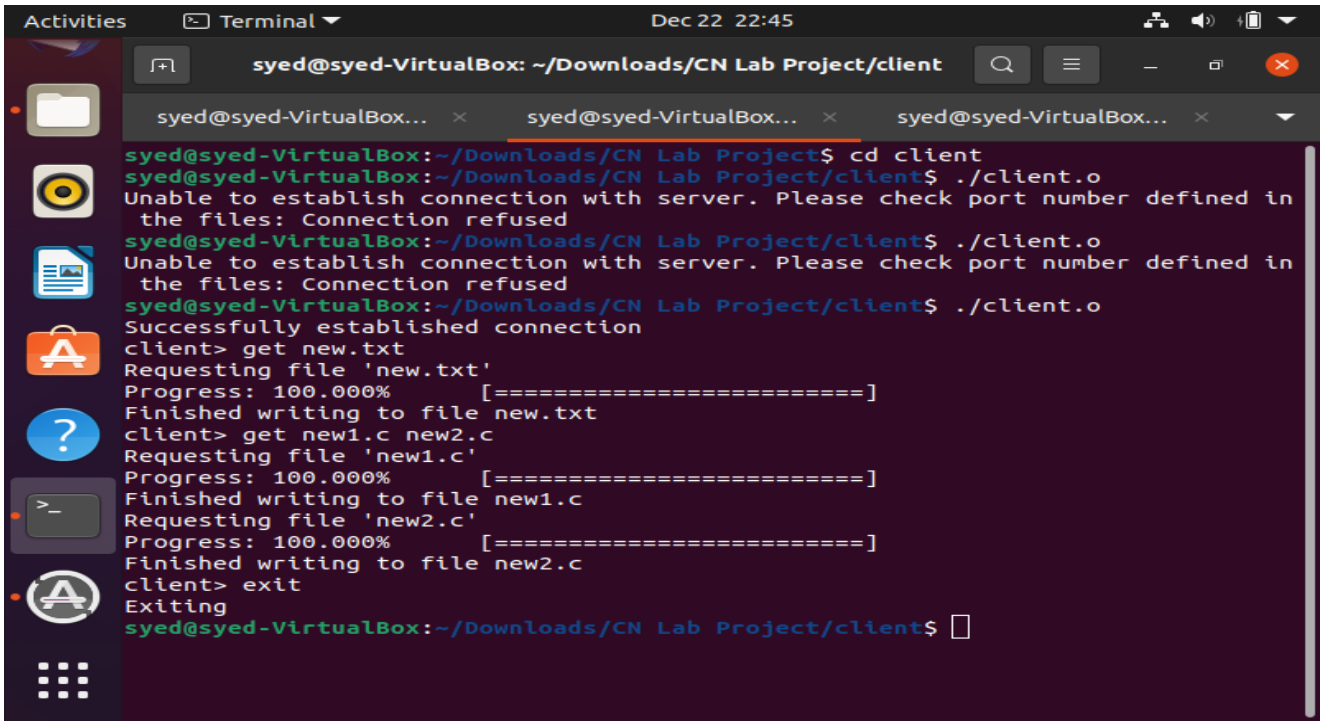

Figure 9: Exit at client side

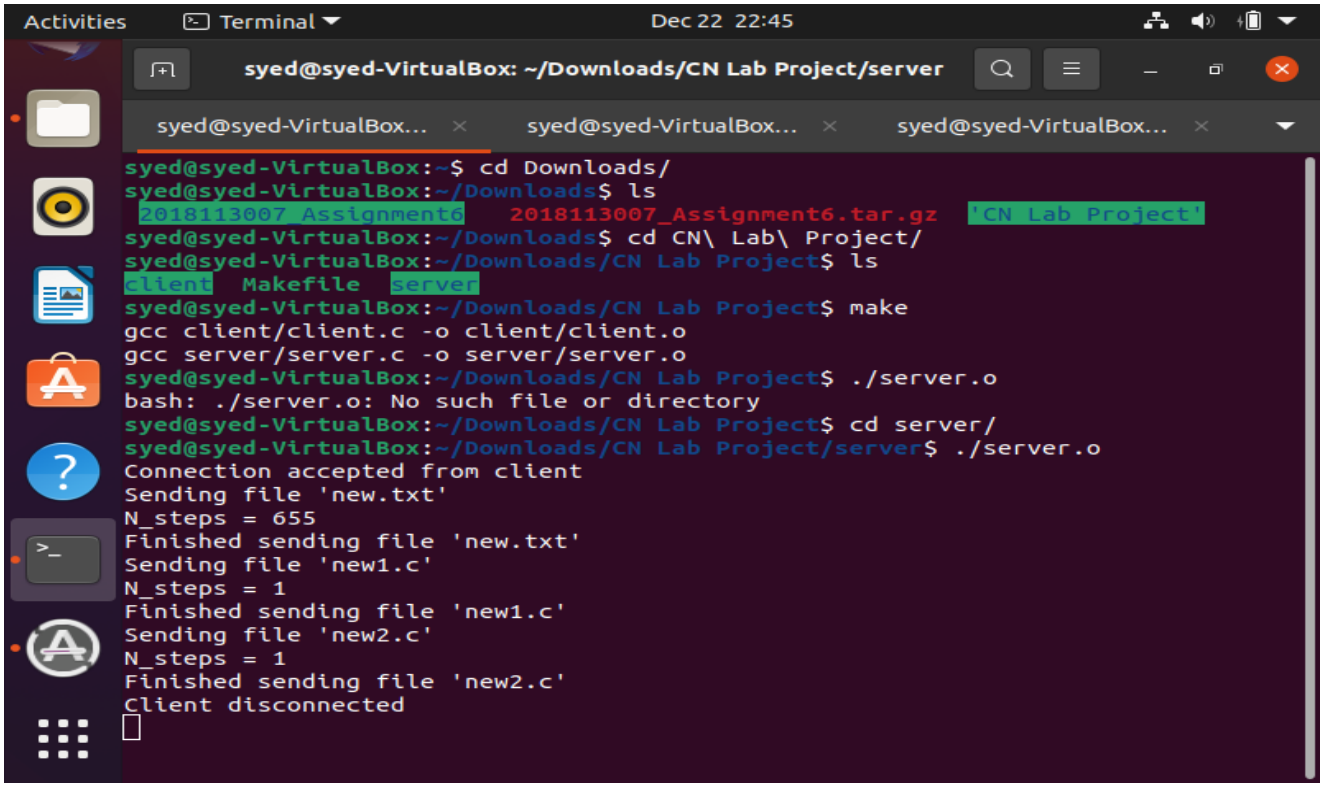

Figure 10: Client disconnected at server side

CONCLUSION AND FUTURE SCOPE

In our project we were successfully able to transfer any kind of file from server to client whether pdf, text, jpeg, etc. We were even able to send multiple files at a time. The file that we are sending got divided into various parts and these parts got transmitted one by one. Each part size was predefined and based on that the number of parts were decided.

Our project explicitly used sockets and ports. You could select the port number you want. This gave us much greater control over the computer network and how we transferred data.

It transferred files in parts and had much greater extensibility. For example, you could use this to transfer files programmatically over the internet and instead of us using the default ports of 80 or 443 for HTTP or HTTPS, we get to choose which port to transfer from

This also allowed us to pause the data transfer. For example, if 3/8 parts are done, then we can pause there and upon resuming, the file transfer will again start from the 4/8th part.

REFERENCES

1. https://www.educba.com/what-is-ddos-attack/
2. https://www.nsnam.org/doxygen/classns3_1_1_on_off_helper.html
3. https://www.geeksforgeeks.org/socket-programming-cc/#:~:text=Socket%20programming%20is%20a%20way,other%20to%20form%20a%20connection.