Ryerson University

Department of Electrical, Computer & Biomedical Engineering
Faculty of Engineering & Architectural Science

COE768 – Computer Networks

LAB REPORT

Semester/Year:	Fall 2024
Lab Number:	Project

Instructor:	Dr. Gul N. Khan
Section No:	062
Submission Date:	2024-11-29
Due Date:	2024-11-29

Student Name	**Student ID**	**Signature**
Carlo Dumlao	501018239	C.D

Table of Contents

1.0 Introduction

By the 1970s (boom of computers), the number of devices in the network had been increasing at a rapid rate. Consequently, tracking them became more complex and fragmented; the size of the IP address uniquely assigned for each device was becoming lengthy, and hard to remember. In fact, in the past, one had to go to the trouble of memorizing those complicated strings of numbers of addresses just to connect to multiple remote devices. It is apparent that there was a need for an automatic system that simplifies the network.

A system called Domain Name System (DNS) was therefore developed to tackle this problem. It is a computer database (or server) containing the IP addresses associated with the names of the websites, or resources. Thanks to this, the user is only required to enter the name of a domain, say google.com, in their device – considering that the IP addresses of web servers are now distributed and handled by a DNS server.

The objective of this project is to explore this client-server architecture. Under the application, there exists an index server and multiple peers. With the index server acting as the DNS in the network, the peers can exchange content to one another. Put simply, by firstly requesting the index (or address) of a content in the index server, a peer can download and share several files from other peers, either a movie, song, or text file. Of course, socket programming was utilized to accomplish the communication between nodes. The index server is bound to a single signal socket which is used to listen and deliver the IP addresses via a UDP protocol. On the other hand, the peer has more than one socket: one for communicating with the index server, and the other sockets are used for sharing contents to other peers through a TCP protocol.

2.0 Description

2.1 Protocols/Sockets

As mentioned before, a peer generates a socket whenever it wishes to make its file available to other peers in the network. Rather than asking the user to assign a unique port number for each created socket – which is quite a tedious task – the TCP module in the OS module is used instead. Figure 2.1 shows the implementation of the TCP module. By assigning the port number to 0, it notifies the TCP to dynamically assign a unique port number of a given socket. This port number is then simply obtained or read through the *getsockname()* call.

```
/* Bind an address to the socket        */
bzero((char *)&server, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(0);
server.sin_addr.s_addr = htonl(INADDR_ANY);
```

Figure 2.1: Dynamically assigning a port to a socket.

In addition, in order to listen to multiple sockets simultaneously, a *select()* call has been implemented. As shown in Figure 2.2, the peer will firstly listen among the collection of existing sockets, including the stdin and the content sockets. If no socket has a pending connection, it will block the program; otherwise, it will immediately proceed and execute the necessary instructions to perform the command requested by the connected peer or server.

```
void waitEvent(){
    FD_ZERO(&afds);
    FD_SET(0, &afds);
    int i;
    for(i = 0; i < files_len; i++) FD_SET(myFiles[i].file_des, &afds);
    memcpy(&rfds, &afds, sizeof(rfds));
    select(FD_SETSIZE, &rfds, NULL, NULL, NULL);
}
```

Figure 2.2: Listening to multiple sockets.

2.2 Index Server

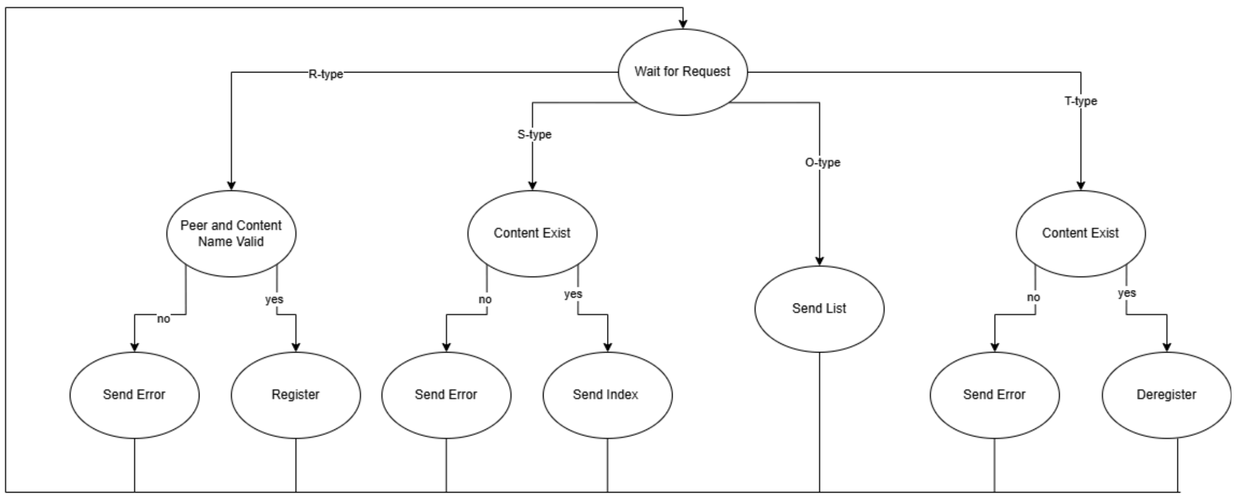


Figure 2.3: State diagram of the index server.

Table 2.1: Description of the states in the index server.

State	Description
Wait for Request	• Checks if there is pending connection in the UDP socket • If there is, it extracts the requested command and parameters (peer name, filename, IP address, and port number) from the 100 bytes PDU packet
Peer and	• Determines if the new content can be registered into the server

Content Name Valid	○ Raises an exception when another peer with the same name has registered the same content
Register	● Registers the new content into the server, assuming it is valid
Content Exist	● Determines if the requested content exist in the server
Send Index	● Sends an index packet to the connected peer ○ Packet consists of the IP address and port number corresponding to the requested content
Send List	● Sends the list of registered contents to the connected peer ○ It will firstly send a packet consisting the number of indexes ○ A sequence of packets are then sent out, containing the actual indexes
Deregister	● De-registers the content requested by the connected peer, assuming that it exists in the table
Send Error	● Sends an error packet to the connected peer ○ When the content to be registered is not valid ○ When the content being requested does not exist in the server

2.3 Peer

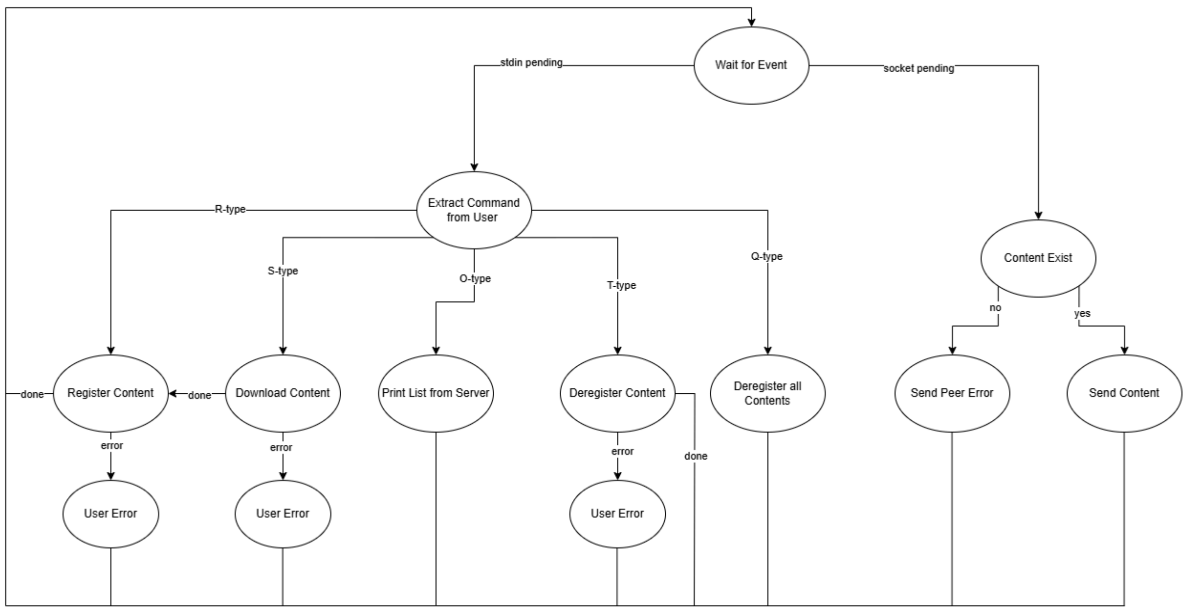


Figure 2.4: State diagram of the peer.

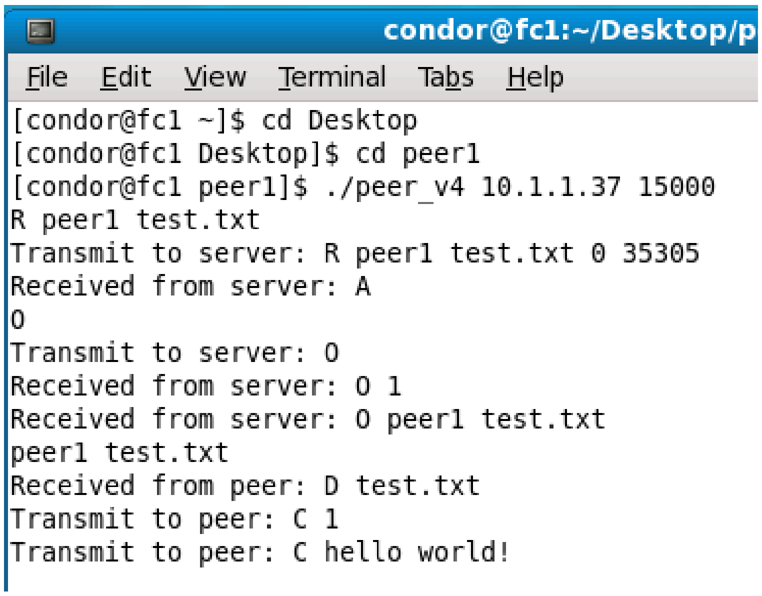
Table 2.2: Description of the states in the peer.

State	Description
Wait for Event	● Waits until one of the sockets, including the stdin and content sockets, has a pending connection
Extract Command for User	● Extracts the command and parameters based on the user input in the terminal
Register Content	● Registers the content into the index server ○ If it has been acknowledged, the state will transition to the Wait for Event indicating a successful registration; otherwise, an exception will be raised
Download Content	● Initially requests the index of the content in the index server ○ If an error packet is received from the index server, it raises an exception ● Once the index has been successfully obtained, it establishes a connection to the content server and sends D-type to trigger the content transfer ○ If an error packet is received from the content server, it raises an exception ● If all goes well, the state will transition to Register Content, allowing itself to be also the content server of that content
Print List from Server	● Sends a list request (O-type) to the index server, and then prints the registered contents into the terminal
Deregister Content	● Deregisters a content in the index server ○ It will raise an exception if it is not been acknowledged
Deregister all Contents	● Deregisters all of its content, by sending a series of T-type PDU to the index server
User Error	● Sends an error message into the terminal when content can not be registered, deregistered, or downloaded.
Content Exist	● Checks whether the file requested by the connected peer exist
Send Content	● Splits the file data into packets, such that each packet has a size of 100 bytes and follows the PDU format ● Sends those series of packets into the connected peer
Send Peer Error	● If there is no such file or content, prompt the connected peer with an error message

3.0 Observation and Analysis

The correctness of the program is presented in the following figures. Inside the network, there is an index server and three peers. Note that they are simulated under a single virtual machine, as a result, the IP addresses of the peers are all identical.

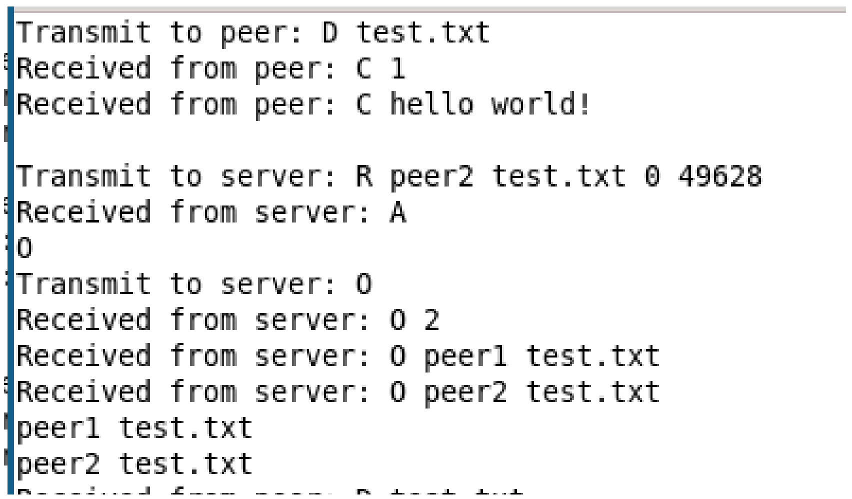
1) In Figure 3.1, peer1 registers a test.txt into the server with a port number of 35305. Notice that when O-type PDU is sent, the content has indeed been successfully registered in the index server. At that moment, a peer – in particular, peer2 – made a request to download that text file.



```
[condor@fc1 ~]$ cd Desktop
[condor@fc1 Desktop]$ cd peer1
[condor@fc1 peer1]$ ./peer_v4 10.1.1.37 15000
R peer1 test.txt
Transmit to server: R peer1 test.txt 0 35305
Received from server: A
O
Transmit to server: O
Received from server: O 1
Received from server: O peer1 test.txt
peer1 test.txt
Received from peer: D test.txt
Transmit to peer: C 1
Transmit to peer: C hello world!
```

Figure 3.1: peer1 terminal.

2) Once peer2 successfully downloaded the test.txt from peer1, it also registered itself as a content server with a port number of 49628. Now, there are two registered test.txt in the index server – that is, one from peer1 and the other from peer2. This is shown in Figure 3.2 below.

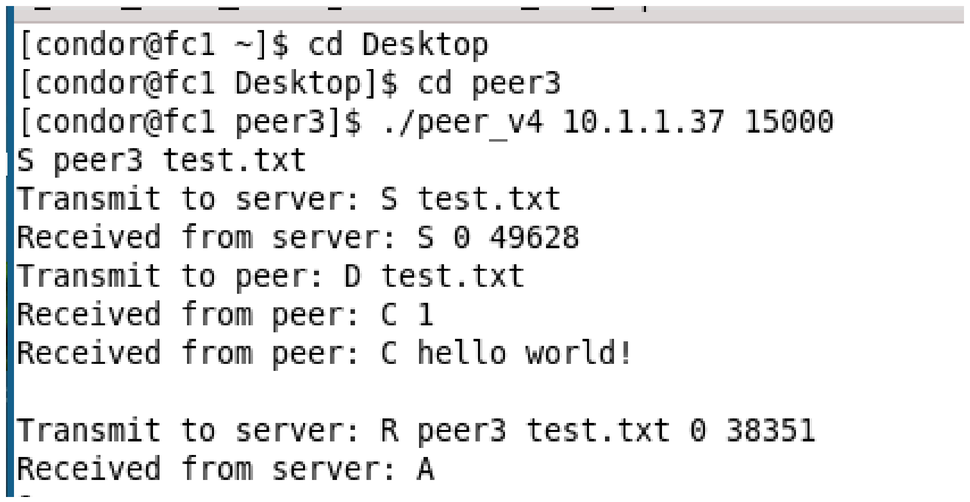


```
Transmit to peer: D test.txt
Received from peer: C 1
Received from peer: C hello world!

Transmit to server: R peer2 test.txt 0 49628
Received from server: A
O
Transmit to server: O
Received from server: O 2
Received from server: O peer1 test.txt
Received from server: O peer2 test.txt
peer1 test.txt
peer2 test.txt
```

Figure 3.2: peer2 terminal.

3) From Figure 3.3, likewise, peer3 requests a content download with the name test.txt. Note that the index server returns the content server owned by peer2, rather than peer1 , to download the text file from. This is to be expected since the index server is structured based on the Least Recently Used (LRU) technique, to ensure that the transfer of files or content are fairly distributed among peers.



```
[condor@fc1 ~]$ cd Desktop
[condor@fc1 Desktop]$ cd peer3
[condor@fc1 peer3]$ ./peer_v4 10.1.1.37 15000
S peer3 test.txt
Transmit to server: S test.txt
Received from server: S 0 49628
Transmit to peer: D test.txt
Received from peer: C 1
Received from peer: C hello world!

Transmit to server: R peer3 test.txt 0 38351
Received from server: A
```

Figure 3.3: peer3 terminal.

4.0 Conclusion

In this project, the UDP and TCP protocol are strictly implemented for server-to-peer and peer-to-peer communication, respectively. The reason behind this stems from efficiency and reliability.

In the server-to-peer, the number of packets being transmitted and received are only miniscule. In a worst case scenario when the index server responds to O-type PDU, the size of the transferred data will be no more than $100(X + 1)\ bytes$, where X is the number of registered contents. Conversely, in a best case scenario when the peer just wants to register, deregister, or access an index of a content, it will only be exactly 100 bytes. Hence, it is much more reasonable to adapt a UDP – a relatively simple protocol – to further improve the performance of the network. Using a complex protocol like TCP that suffers a higher overhead is unnecessary, considering the data length is so small that the chance of it being corrupted during the transmission time is quite unlikely.

On the other hand, in peer-to-peer, the number of packets can be huge. From a user perspective, the program is expected to transfer files ranging from Megabytes to Gigabytes. Therefore, UDP is not suitable for this job – although delivery is fast, the data is now susceptible to errors during a transmission. A TCP is best suited in this case, since it guarantees that the large content is correctly transferred between peers.

5.0 Appendix

5.1 Server

```
#include <stdbool.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include <time.h>

#define PEER_MAX 10
#define CONTENT_MAX 10
#define ADDR_MAX 10

//different commands
#define R 0
#define S 1
#define O 2
#define T 3

//States
#define WAITPEER 0
#define CHECKREG 1
#define ERROR 2
#define REG 3
#define CONTENTEXIST_S 4
#define SENDADDR 5
#define SENDLIST 6
#define CONTENTEXIST_T 7
#define DEREG 8

//Table to store the indexes
struct Table{
    char peer[PEER_MAX];
    char content[CONTENT_MAX];
    char addr[ADDR_MAX];
    char sock[10];
    int recent;
```

```
} myTable[20];
int table_len = 0;

//PDU
struct PDU{
    int command;
    char peer[PEER_MAX];
    char content[CONTENT_MAX];
    char addr[ADDR_MAX];
    char sock[10];
} myPDU;


struct  sockaddr_in fsin;   /* the from address of a client */
char    *pts;
int   sock;                 /* server socket             */
int   alen;                 /* from-address length             */
struct  sockaddr_in sin;  /* an Internet endpoint address          */
int     s, type;            /* socket descriptor and socket type    */
int   port=3000;


int state = WAITPEER;

//Create the UDP socket
void initServer(int argc, char *argv[]){
    switch(argc){
            case 1:
                break;
            case 2:
                port = atoi(argv[1]);
                break;
            default:
                fprintf(stderr, "Usage: %s [port]\n", argv[0]);
                exit(1);
      }

    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port);

    /* Allocate a socket */
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0)
```

```
      fprintf(stderr, "can't creat socket\n");

    /* Bind the socket */
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      fprintf(stderr, "can't bind to %d port\n",port);
    listen(s, 5);
      alen = sizeof(fsin);
}

void replaceChar(char text[], char old, char new){
    int i;
    for(i = 0; text[i] != '\0'; i++){
        if(text[i] == old) text[i] = new;
    }
}

//Read the packet from a connected peer
void getPeerInput(){
    int v = 2;
    char data[100];

    if (recvfrom(s, data, 100, 0,
      (struct sockaddr *)&fsin, &alen) < 0)
    fprintf(stderr, "recvfrom error\n");

    printf("Received: ");
    printf(data);
    printf("\n");

    replaceChar(data, ' ', '\0');

    if(strcmp(data, "R") == 0){
        myPDU.command = R;

        strcpy(myPDU.peer, &data[v]);

        while(data[v++] != '\0');

        strcpy(myPDU.content, &data[v]);

        while(data[v++] != '\0');

        strcpy(myPDU.addr, &data[v]);

        while(data[v++] != '\0');
        strcpy(myPDU.sock, &data[v]);

    }
    else if(strcmp(data, "S") == 0){
        myPDU.command = S;

        strcpy(myPDU.content, &data[v]);
    }
    else if(strcmp(data, "O") == 0){
        myPDU.command = O;

    }
    else if(strcmp(data, "T") == 0){
        myPDU.command = T;

        strcpy(myPDU.peer, &data[v]);

        while(data[v++] != '\0');

        strcpy(myPDU.content, &data[v]);
    }

}

//Check if the content to be registered is valid
bool samePeerContent(char peer[], char content[]){
    int i;
    for(i = 0; i < table_len; i++){
        if(strcmp(peer, myTable[i].peer) == 0 && strcmp(content,
myTable[i].content) == 0) return true;
    }
    return false;
}

//Check if a given content  exist in the table
bool contentExist(char content[]){
    int i;
    for(i = 0; i < table_len; i++){
        if(strcmp(content, myTable[i].content) == 0) return true;
    }
    return false;
}

//Check if a given peer exist in the table
```

```
bool contentPeerExist(char peer[], char content[]){
    int i;
    for(i = 0; i < table_len; i++){
        if(strcmp(content, myTable[i].content) == 0 && strcmp(peer,
myTable[i].peer) == 0) return true;
    }
    return false;
}

//Find the least recently used index
int findContentLeastUsed(char content[]){
    int val = 999;
    int index = -1;
    int i;
    for(i = 0; i < table_len; i++){
        if(strcmp(content, myTable[i].content) == 0 &&
myTable[i].recent < val){
            val = myTable[i].recent;
            index = i;
        }
    }
    return index;
}


//Register a content
void appendContent(char peer[], char content[], char addr[], char
sock[]){
    strcpy(myTable[table_len].peer, peer);
    strcpy(myTable[table_len].content, content);
    strcpy(myTable[table_len].addr, addr);
    strcpy(myTable[table_len].sock, sock);
    myTable[table_len].recent = 0;
    table_len++;
}

//Deregister a content
void deleteContent(char peer[], char content[]){
    int i;
    int v;
    for(i = 0; i < table_len; i++){
        if(strcmp(peer, myTable[i].peer) == 0 && strcmp(content,
myTable[i].content) == 0){
            for(v = i; v < table_len - 1; v++){
                strcpy(myTable[v].peer, myTable[v+1].peer);
                strcpy(myTable[v].content, myTable[v+1].content);
                strcpy(myTable[v].addr, myTable[v+1].addr);
                strcpy(myTable[v].sock, myTable[v+1].sock);
                myTable[v].recent = myTable[v+1].recent;
            }
            table_len--;
            return;
        }
    }
}

//Send an error PDU
void sendE(char error[]){
    char data[100];
    strcpy(data, "E ");
    strcat(data, error);

    (void) sendto(s, data, 100, 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

    printf("Transmit: ");
    printf(data);
    printf("\n");
}

//Send an acknowledge PDU
void sendA(){
    char data[100];
    strcpy(data, "A");

     (void) sendto(s, data, 100, 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

    printf("Transmit: ");
    printf(data);
    printf("\n");
}

//Send the index of the requested content
void sendS(int i){
    myTable[i].recent++;

    char data[100];
    strcpy(data, "S ");
    strcat(data, myTable[i].addr);
    strcat(data, " ");
    strcat(data, myTable[i].sock);

     (void) sendto(s, data, 100, 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

    printf("Transmit: ");
    printf(data);
    printf("\n");
}

//Send the registered conents
void sendO(){
    char data[100];
    strcpy(data, "O ");

    char temp[10];
    sprintf(temp, "%d", table_len);
    strcat(data, temp);

     (void) sendto(s, data, 100, 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

    printf("Transmit: ");
    printf(data);
    printf("\n");

    int i;
    for(i = 0; i < table_len; i++){
        strcpy(data, "O ");
        strcat(&data[2],  &myTable[i].peer[0]);
        strcat(&data[2], " ");
        strcat(&data[2], &myTable[i].content[0]);

         (void) sendto(s, data, 100, 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

        printf("Transmit: ");
        printf(data);
        printf("\n");
    }
}

int main(int argc, char *argv[]){
    initServer(argc, argv);
    char error[50];

    while(true){
        switch(state){
            //Wait if the UDP socket has a pending connection
            case WAITPEER:
                getPeerInput();
                if(myPDU.command == R) state = CHECKREG;
                else if(myPDU.command == S) state = CONTENTEXIST_S;
                else if(myPDU.command == O) state = SENDLIST;
                else if(myPDU.command == T) state = CONTENTEXIST_T;
                break;
            //Check if the content from a connected peer can be
registered
            case CHECKREG:
                if(samePeerContent(myPDU.peer, myPDU.content)){
                    strcpy(error, "Error: Same peer and content");
                    state = ERROR;
                }
                else state = REG;
                break;
            //Prompt an error to the connected peer
            case ERROR:
                sendE(error);
                state = WAITPEER;
                break;
            //Register the content
            case REG:
                appendContent(myPDU.peer, myPDU.content, myPDU.addr,
myPDU.sock);
                sendA();
                state = WAITPEER;
                break;
            //Check if the requested content exist
            case CONTENTEXIST_S:
                if(!contentExist(myPDU.content)){
                    strcpy(error, "Error: Content does not exist");
                    state = ERROR;
                }
                else state = SENDADDR;
                break;
            //Send index od the requested content
            case SENDADDR:
                sendS(findContentLeastUsed(myPDU.content));
                state = WAITPEER;
                break;
            //Send the registered index
            case SENDLIST:
                sendO();
                state = WAITPEER;
                break;
            //Prompt an error to the connected peer
            case CONTENTEXIST_T:
                if(!contentPeerExist(myPDU.peer, myPDU.content)){
                    strcpy(error,  "Error: Peer and Content does not
exist");
                    state = ERROR;
                }
                else  state = DEREG;
                break;
            //Deregister a conent
            case DEREG:
                deleteContent(myPDU.peer, myPDU.content);
                sendA();
                state = WAITPEER;
                break;
        }
    }





    return 0;
}
```

5.2 Peer

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

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
#include <sys/signal.h>
#include <sys/wait.h>

#define PEER_MAX 10
#define CONTENT_MAX 10
#define FILES_MAX 10
#define ADDR_MAX 10

//different commands
#define R 0
#define S 1
#define O 2
#define T 3
#define Q 4
#define A 5
#define E 6
#define D 7
#define C 8

//States
#define WAIT_EVENT 0
#define PEER_REQ 1
#define SEND_FILE 2
#define PEER_ERROR 3
#define USER_INPUT 4
#define R_TYPE 5
#define USER_ERROR 6
#define S_TYPE 7
#define O_TYPE 8
#define T_TYPE 9
#define Q_TYPE 10

//PDU for holding the user input
struct PDU{
    int command;
    char peer[PEER_MAX];
    char content[CONTENT_MAX];
    char addr[ADDR_MAX];
    char sock[10];
} myPDU;

//PDU for holding the server and peer communication
```

```
struct PDU_v2{
    int command;
    char data[100];
} serverPDU, peerPDU;


//My content servers
struct Files{
    char peer[PEER_MAX];
    char content[CONTENT_MAX];
    char addr[ADDR_MAX];
    char sock[10];
    int file_des;
} myFiles[FILES_MAX];

int files_len = 0;

fd_set rfds, afds;

int server_sock;

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


//Listen for any pending connections in the socket
void waitEvent(){
    FD_ZERO(&afds);
    FD_SET(0, &afds);
    int i;
    for(i = 0; i < files_len; i++) FD_SET(myFiles[i].file_des,
&afds);
    memcpy(&rfds, &afds, sizeof(rfds));
    select(FD_SETSIZE, &rfds, NULL, NULL, NULL);
}


void replaceChar(char text[], char old, char new){
    int i;
    for(i = 0; text[i] != '\0'; i++){
        if(text[i] == old) text[i] = new;
    }
}

//Extract the command and parammeters from the stdin
bool extractUserCmd(char text[]){
    char to_test[30];
    to_test[0] = text[0];
    to_test[1] = '\0';

    int i;
    for(i = 0; text[i] != '\0'; i++){
        if(text[i] == ' '){
            strcat(to_test, " ");
        }
    }


    i = 2;
    if(strcmp(to_test, "R  ") == 0 && strcmp(text, "R  ") != 0){
        replaceChar(text, ' ', '\0');

        myPDU.command = R;
        strcpy(myPDU.peer, &text[i]);
        while(text[i++] != '\0');
        strcpy(myPDU.content, &text[i]);
        return true;
    }
    else if(strcmp(to_test, "S  ") == 0 && strcmp(text, "S  ") != 0){
        replaceChar(text, ' ', '\0');

        myPDU.command = S;
        strcpy(myPDU.peer, &text[i]);
        while(text[i++] != '\0');
        strcpy(myPDU.content, &text[i]);
        return true;
    }
    else if(strcmp(to_test, "O") == 0){
        myPDU.command = O;
        return true;
    }
    else if(strcmp(to_test, "T  ") == 0 && strcmp(text, "T  ") != 0){
        replaceChar(text, ' ', '\0');

        myPDU.command = T;
        strcpy(myPDU.peer, &text[i]);
        while(text[i++] != '\0');
        strcpy(myPDU.content, &text[i]);
        return true;
    }
    else if(strcmp(to_test, "Q") == 0){
        myPDU.command = Q;
        return true;
    }
    return false;
}

//Read the PDU packet from the index server
void getServer(){
    char text[100];

    read(server_sock, text, 100);
    printf("Received from server: ");
    printf(text);
    printf("\n");

    switch(text[0]){
        case 'A':
            serverPDU.command = A;
            break;
        case 'E':
            serverPDU.command = E;
            strcpy(serverPDU.data, &text[2]);
            break;
        case 'O':
            serverPDU.command = O;
            strcpy(serverPDU.data, &text[2]);
            break;
        case 'S':
            serverPDU.command = S;
            strcpy(serverPDU.data, &text[2]);
            break;
    }
}

//Send a PDU packet to the server
void sendServer(int cmd, char p[], char c[], char addr[], char
sock[]){
    char data[100] = "";
    switch(cmd){
        case R:
            strcpy(data, "R ");
            strcat(data, p);
            strcat(data, " ");
            strcat(data, c);
            strcat(data, " ");
            strcat(data, addr);
            strcat(data, " ");
            strcat(data, sock);
            break;
        case S:
            strcpy(data, "S ");
            strcat(data, c);
            break;
        case O:
            strcpy(data, "O");
            break;
        case T:
            strcpy(data, "T ");
            strcat(data, p);
            strcat(data, " ");
            strcat(data, c);
            break;
    }

    write(server_sock, data, 100);

    printf("Transmit to server: ");
    printf(data);
    printf("\n");
}

//Get the registered contents from the index server
void getServerList(char text[], int len){
    int i;
    for(i = 0; i < len; i++){
        getServer();
        strcat(text, serverPDU.data);
        strcat(text, "\n");
    }
}

//Send a packet to the connected peer
void sendPeer(int cmd, char text[], int socket){
    char data[100];
    switch(cmd){
        case E:
            strcpy(data, "E ");
            strcat(data, text);
            break;
        case D:
            strcpy(data, "D ");
            strcat(data, text);
            break;
        case C:
            strcpy(data, "C ");
            strcat(data, text);
            break;

    }
    write(socket, data, 100);

    printf("Transmit to peer: ");
    printf(data);
    printf("\n");
}

//Read the packet from the connected peer
void getPeer(int socket){
    char text[100];
    read(socket, text, 100);

    printf("Received from peer: ");
    printf(text);
    printf("\n");

    switch(text[0]){
        case 'E':
            peerPDU.command = E;
            strcpy(peerPDU.data, &text[2]);
            break;
        case 'D':
            peerPDU.command = D;
            strcpy(peerPDU.data, &text[2]);
            break;
        case 'C':
            peerPDU.command = C;
            strcpy(peerPDU.data, &text[2]);
            break;

    }
```

```
}

//Get the size of a given file
int getFileLength(char filename[]){
    FILE* fp;
    int count = 0;
    char c;
    fp = fopen(filename, "r");
    for (c = getc(fp); c != EOF; c = getc(fp)) count = count + 1;
    fclose(fp);
    return count;
}

//Read the incoming file from the content server
void getPeerFile(char filename[], int len, int socket){
    FILE *fptr;
    fptr = fopen(filename, "a");
    int i;
    for(i = 0; i < len; i++){
        getPeer(socket);                //Example
        fprintf(fptr, peerPDU.data);
    }

    fclose(fptr);
}

//Send file to the connected peer
void sendPeerFile(char filename[], int socket){
    char data[100];
    strcpy(data, "C ");

    int len = getFileLength(filename)/97 + 1;

    FILE *fptr;
    fptr = fopen(filename, "r");
    int i;
    for(i = 0; i < len; i++){
        fgets(&data[2], 98, fptr);

        write(socket, data, 100);

        printf("Transmit to peer: ");
        printf(data);
        printf("\n");
    }
    fclose(fptr);
}

//Check if the file exist
bool fileExist(char filename[]){
    FILE* fp;
    fp = fopen(filename, "r");
    bool exist = true;
    if (fp == NULL) exist = false;
    else fclose(fp);
    return exist;
}

//Create a new TCP socket for a given file
void createFileSocket(char peer[], char content[]){
    int    sd;
     struct     sockaddr_in server, reg_addr;

     /* 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(0);
     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);

    int alen = sizeof (struct sockaddr_in);
    getsockname(sd, (struct sockaddr *) &reg_addr, &alen);

    myFiles[files_len].file_des = sd;
    sprintf(myPDU.sock, "%d", reg_addr.sin_port);
    sprintf(myPDU.addr, "%d", server.sin_addr.s_addr);
    strcpy(myFiles[files_len].peer, peer);
    strcpy(myFiles[files_len].content, content);
    strcpy(myFiles[files_len].addr, myPDU.addr);
    strcpy(myFiles[files_len].sock, myPDU.sock);
    files_len++;
}

//Close an existing TCP socket
void destroyFileSocket(char peer[], char content[]){
    int i;
    int j;
    for(i = 0; i < files_len; i++){
        if(strcmp(myFiles[i].peer, peer) == 0 &&
strcmp(myFiles[i].content, content) == 0 ){
            for(j = i; j < files_len-1; j++){
                close(myFiles[j].file_des);
                strcpy(myFiles[j].peer, myFiles[j+1].peer);
                strcpy(myFiles[j].content, myFiles[j+1].content);
                strcpy(myFiles[j].addr, myFiles[j+1].addr);
                strcpy(myFiles[j].sock, myFiles[j+1].sock);
          myFiles[j].file_des = myFiles[j+1].file_des;
            }
        }
    }
    files_len--;
}

//Close the recently created socket
void destroyRecentFileSocket(){
  files_len--;
  close(myFiles[files_len].file_des);
}

//Establish a connection to the server
void connectServer(int argc, char **argv){
    char    *host = "localhost";
     int   port = 3000;
     struct hostent   *phe; /* pointer to host information entry */
     struct sockaddr_in sin;     /* an Internet endpoint address
*/
     int   s, n, type;       /* socket descriptor and socket type */

     switch (argc) {
     case 1:
```

```
                break;
        case 2:
                host = argv[1];
        case 3:
                host = argv[1];
                port = atoi(argv[2]);
                break;
        default:
                fprintf(stderr, "usage: UDPtime [host [port]]\n");
                exit(1);
        }

        memset(&sin, 0, sizeof(sin));
            sin.sin_family = AF_INET;
            sin.sin_port = htons(port);

      /* Map host name to IP address, allowing for dotted decimal */
            if ( phe = gethostbyname(host) ){
                        memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
            }
            else if ( (sin.sin_addr.s_addr = inet_addr(host)) ==
INADDR_NONE )
                fprintf(stderr, "Can't get host entry \n");

      /* Allocate a socket */
            s = socket(AF_INET, SOCK_DGRAM, 0);
            if (s < 0)
                fprintf(stderr, "Can't create socket \n");


      /* Connect the socket */
            if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
                fprintf(stderr, "Can't connect to %s\n", host);

      server_sock = s;
}

//Establish a a peer-to-peer connection
int connectPeerServer(char addr[], char sock[]){
        int   sd, port;
        struct      hostent           *hp;
        struct      sockaddr_in server;
        int host;

            host = atoi(addr);
            port = atoi(sock);

        /* 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 = port;
            server.sin_addr.s_addr = host;

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

      return sd;
}

int main(int argc, char **argv){
      connectServer(argc, argv);

      char error[50];
      int state = WAIT_EVENT;
      char file_length[10];

      char user_input[30];
      char myList[150] = "";

      int temp_sock;
      struct sockaddr_in client;
      int client_len;
      int j;
      int i;

      while(true){
            switch(state){
               //Wait until one of the sockets has a pending connection
               case WAIT_EVENT:
                        waitEvent();
                        if(FD_ISSET(0,&rfds)) state = USER_INPUT;
```

```
                else{
                    for(i = 0; i < files_len; i++){
                        temp_sock = myFiles[i].file_des;
                        if(FD_ISSET(temp_sock, &rfds)){
                                client_len = sizeof(client);
                            temp_sock = accept(temp_sock, (struct
sockaddr *)&client, &client_len);
                            state = PEER_REQ;
                            break;
                        }
                    }
                }
                break;
            //Read the command from connected peer
            case PEER_REQ:
                getPeer(temp_sock);
                if(fileExist(peerPDU.data)) state = SEND_FILE;
                else{
                    state = PEER_ERROR;
                    strcpy(error, "File does not exist");
                }
                break;
            //if the requested file exist, send the file to the
connected peer
            case SEND_FILE:
                sprintf(file_length, "%d",
getFileLength(peerPDU.data)/97 + 1);
                sendPeer(C, file_length, temp_sock);
                sendPeerFile(peerPDU.data, temp_sock);
                close(temp_sock);
                state = WAIT_EVENT;
                break;
            //Otherwise, prompt the connected peer with an error
message
            case PEER_ERROR:
                sendPeer(E, error, temp_sock);
                close(temp_sock);
                state = WAIT_EVENT;
                break;
            //Extract the user input in the terminal
            case USER_INPUT:
                fgets(user_input, 30, stdin);
                user_input[strcspn(user_input, "\n")] = 0;
                if(!extractUserCmd(user_input)){
                    state = USER_ERROR;
                    strcpy(error, "Command is invalid");
                }
                else if(myPDU.command == R) state = R_TYPE;
                else if(myPDU.command == S) state = S_TYPE;
                else if(myPDU.command == T) state = T_TYPE;
                else if(myPDU.command == O) state = O_TYPE;
                else if(myPDU.command == Q) state = Q_TYPE;
                break;
            //Register the content
            case R_TYPE:
                createFileSocket(myPDU.peer, myPDU.content);
                sendServer(R, myPDU.peer, myPDU.content, myPDU.addr,
myPDU.sock);
                getServer();
                if(serverPDU.command == A) state = WAIT_EVENT;
                else if(serverPDU.command == E){
                    state = USER_ERROR;
                    strcpy(error, serverPDU.data);
                    destroyRecentFileSocket();
                }
                break;
            //Print an error into the terminal
            case USER_ERROR:
                printf(error);
                printf("\n");
                state = WAIT_EVENT;
                break;
            //Download a content
            case S_TYPE:
                sendServer(S, NULL, myPDU.content, NULL, NULL);
                getServer();
                if(serverPDU.command == E){
                    state = USER_ERROR;
                    strcpy(error, serverPDU.data);
                }
                else if(serverPDU.command  == S){
                    j = 0;
                    while(serverPDU.data[j++] != ' ');
                    serverPDU.data[j-1] = '\0';
                    temp_sock = connectPeerServer(serverPDU.data,
&serverPDU.data[j]);

                    sendPeer(D, myPDU.content, temp_sock);
                    getPeer(temp_sock);
                    if(peerPDU.command == E){
                        close(temp_sock);
                        state = USER_ERROR;
                        strcpy(error, peerPDU.data);
                    }
                    else if(peerPDU.command == C){
                        getPeerFile(myPDU.content,
atoi(peerPDU.data), temp_sock);
                        close(temp_sock);
                        state = R_TYPE;
                    }
                }
                break;
            //Print the registered contents
            case O_TYPE:
                myList[0] = '\0';
                sendServer(O, NULL, NULL, NULL, NULL);
                getServer();
                getServerList(myList, atoi(serverPDU.data));
                printf(myList);
                state = WAIT_EVENT;
                break;
            //Deregister a content
            case T_TYPE:
                sendServer(T, myPDU.peer, myPDU.content, NULL, NULL);
                getServer();
                if(serverPDU.command == A) state = WAIT_EVENT;
                else if(serverPDU.command == E){
                    state = USER_ERROR;
                    strcpy(error, serverPDU.data);
                }
                break;
            //Deregister all owned content servers
            case Q_TYPE:
                while(files_len > 0){
                    sendServer(T, myFiles[0].peer,
myFiles[0].content, NULL, NULL);
                    destroyFileSocket(myFiles[0].peer,
myFiles[0].content);
                }
                return 0;

        }
    }
}
```