Toronto Metropolitan University

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

Course Title:	Computer Networks
Course Number:	COE768
Semester/Year (e.g. F2017)	F2023

Instructor	Dr. Truman Yang

Assignment/Lab Number:	N/A
Assignment/Lab Title:	COE768 Project: Peer-to-Peer Application

Submission Date:	12/01/2023
Due Date:	12/01/2023

Student LAST Name	**Student FIRST Name**	**Student Number**	**Section**	**Signature***
Rahman	Tamim	500967494	3	TR
Madhu	Yadu Krishnan	500975010	3	Y.M

COE768 Project: Peer-to-Peer Application

Introduction

Our final project for COE 768 is building out a peer to peer application. Building the application requires solid knowledge of networking concepts and applying them thoroughly. The peer-to-peer application implements a basic networking structure where multiple peers can communicate with each other. With the help of a centralized server that acts as a registry for content, peers can seek out other peers for content available within the network by fetching all the available content at its location.

The centralized server, in this case, acts as a registry, as mentioned above, where the registry holds specific entry information for each piece of content available within the network. Each entry will specifically contain its content name (a piece of content fellow peers can access), content address (the IP address of this piece of content), content type and server name (peer name). Based on the entries in the index server (central server), peers can either register content based on the parameters mentioned above or request to download any content that matches one of the existing entries.

To establish a connection between peers or servers, knowledge of concepts regarding socket programming is necessary. Our design will incorporate TCP and UDP protocols for establishing connections between fellow peers and the index server. More specifically, the UDP protocol is utilized for establishing connections between peers and the index server. In contrast, the TCP protocol is utilized for establishing connections between clients and peers serving content. UDP protocol is suited for our use case because of its simplicity in establishing the connection between clients and the index server. TCP is a reliable protocol which handles lost packets ensuring lossless transmission of data, which is essential for downloading content.

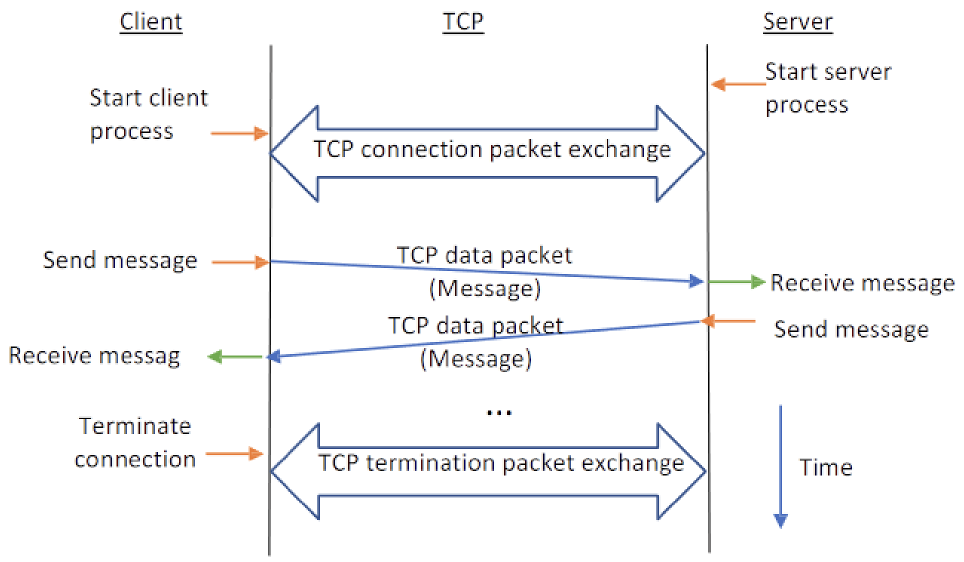

Figure 1 : TCP Protocol

Figure 1 above shows how clients typically establish a TCP connection, transfer data and terminate. It is important to note that TCP requires a 3-way handshake protocol to establish connections and acknowledgement packs for each data transfer to ensure lossless data transfers as well.

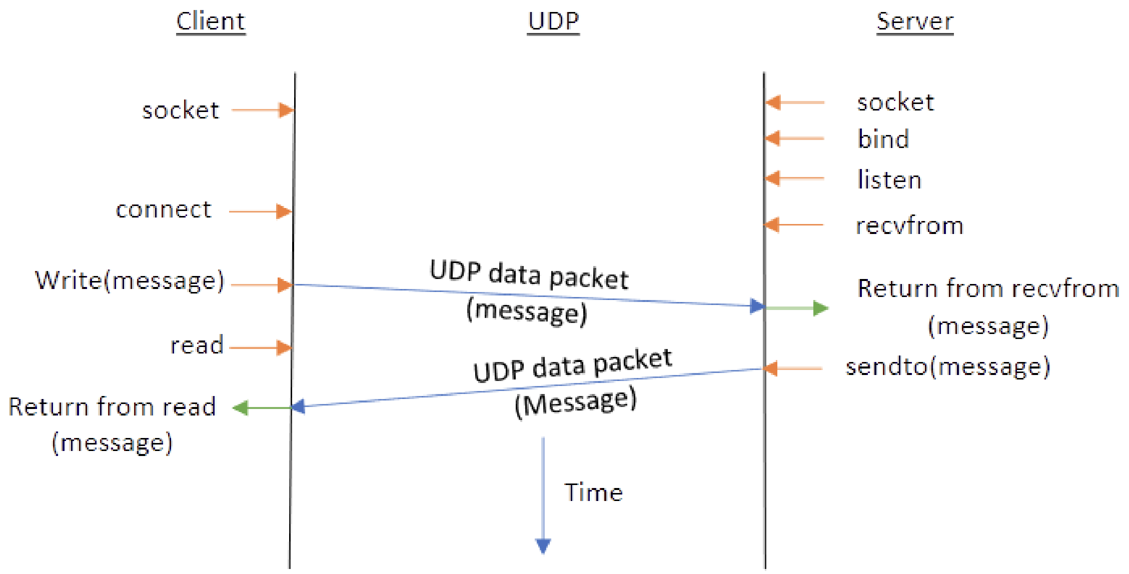
Client
UDP
Server
socket
connect
Write(message)
read
Return from read (message)
UDP data packet (message)
UDP data packet (Message)
Time
socket
bind
listen
recvfrom
Return from recvfrom (message)
sendto(message)

Figure 2 : UDP Protocol

Figure 2 above shows how clients establish a UDP connection. In our case, we utilize UDP to connect with the index server.

Program Descriptions

Server Application

Our application has a main server which has a registry that contains a list of all downloadable contents, its address and other related information. We call this main server the index server. A basic approach was developed based on the requirements of the server, where it had to:

- Implement communication with other clients through UDP.
- Implement a registry to store information regarding downloading content. The information to store includes
    - Type of content.
    - Peer name.
    - Content name.
    - Address
- Send and receive commands from the clients that include:
    - Request to register content.
    - Request to de-register content.
    - Request to search for available content.

Based on these requirements, we developed the server to establish a connection based on the UDP protocol. We also had to implement a data packet structure called PDU with special fields denoting the packet type. This is necessary to distinguish different types of data packets. The PDU types related to the index server include:

- Type ‘R’ (Content Registration): This packet is sent by the client to indicate a request to register content in the index server.
- Type ‘S’ (Search Content): This packet is sent by the client to indicate a request to search for the content present in the data field of this packet. The response depends on whether the request is valid, meaning whether the request data is found in the registry. Otherwise, an error message will be sent back. More details about the error message can be found below.
- Type ‘T’ (Content De-registration): The client sends this packet to indicate a request to de-register its content found in the index server.
- Type ‘O’ (List of On-Line Registered Content): This packet is sent by a client as a request to list all available downloadable content from the index server. The index server responds by sending a list.
- Type ‘A’ (Acknowledgement): This is a simple acknowledge packet sent from the server to the peer.
- Type ‘E’ (Error): This is an error packet sent from the server to the error to indicate an invalid request.

Our approach and the final index server implementation are completely based on the above-listed requirements and PDU structure.

The index server is run by executing its executable and passing a port number as a parameter.
Once the index server is up and running, it is set to run indefinitely, where it listens for incoming UDP requests. Once a client connects with the server, the server will attempt to read the PDU type of the data packet sent by the client. Based on the PDU types listed above, the client may choose to either register content, search for content or deregister its content. Suppose the client chooses to register content. In that case, the server will check its registry to match the client's content request. If a match occurs, an 'E' error packet will be sent with an appended message indicating the client to return the request with a different client name or content name. If there is no match, the associated content will be registered along with the client/peer address information within the registry of the index server. Subsequently, it will send back an 'A' acknowledge packet to indicate successful content registration. Suppose another client wants a list of registered content. The client can send a request of the PDU type 'O'. Once the server receives the packet, it simply calls its function to search and retrieve the list of available content and related information. Since each entry stores the "Content Name", and "Peer Name" interchangeably mentioned as Client Name in the document, and "Content Address", these are displayed as a result. Finally, the client can request to de-register its content. The packet to send should be of type 'T'. Once the server receives this packet, it searches for entries with the associated peer name and removes it from the registry. An 'A' type packet is sent if successfully removed; an 'E' type packet is sent back to indicate an error which happens if the client never had any registered content in the server.

Client Application

The basic breakdown of our approach with the client program is that we used a prompt-by-prompt method of soliciting information from the user. This allowed us to have a quick and easy to gather information from the user instead of maintaining input loops. Since it was an easy implementation and still within the project's requirements, we decided to stay with this approach. The program runs the executable and passes in the index server address and port information. After the executable is run, the program will prompt the user to select whether the user is trying to become a content-serving peer or a content-receiving peer. They can choose by entering the corresponding number in the terminal prompt. Depending on their input, the program's behaviour will transform to fit the user's needs.

If they choose to be a content-serving peer, they are prompted to provide information regarding themselves and the content they want to serve. For the content, all that is needed is the name, and the peer needs to provide its IP address and its desired peer name. The content name and the peer name must be unique, as it is an issue for the index server if there are overlapping names in those information domains. If the user selects to be a content-receiving peer, they will be prompted to provide the name of the content they would like to download and the name of the content-serving peer they want to download from. A future improvement to this application would be to communicate with the index server and have the user select from a list. After the download information is provided to the program, it communicates with the content-serving peer to download the content. Upon receiving the file, the program can serve the content to other peers because it now has it. Like creating a content-serving peer, the user will be prompted to provide the peer's name and address. The user is shown which port the content is being hosted on. That is an overview of the program flow from the perspective of a client user.

Observation and Analysis

During our demonstration of the application, we set up 3 peer or client instances by the name: “Peer 1”, “Peer 2”, and “Peer 3”. “Peer 1” was tasked with registering content to the server. The content was a simple text file named “text1.txt,” which contained dummy text. On successful completion, the client received an acknowledged packet. “Peer 2” now attempts to download the content registered by “Peer 1”. It connects to the server and sends a search request with the content name registered by “Peer 1”. The server returns the address and port number associated with the requested content. With the information needed to connect with “Peer 1”, “Peer 2” makes a TCP connection request with “Peer 1” to download the content. Once the download is finished, “Peer 2” should also act as a content server for that specific content. We can test this by requesting “Peer 3” to request the index server to list all the available content. The server prints the available content list, which marks “Peer 1” and “Peer 2” and serves the same content. “Peer 3” now attempts to download “text1.txt,” and the index server returns the request with a response containing the address and port number of “Peer 2” now. “Peer 3” now establishes the TCP connection with “Peer 2” and downloads the file, which indicates that the client-server transfer was indeed successful. All the peers now quit, sending a de-registration request to the server, which then updates the registry within the server. The server now has no content.

Based on the results of the demonstration above, it is conclusive that the application adheres to the requirements of this project.

Conclusions

This peer-to-peer socket programming project showcased the power and efficiency of a decentralized content distribution network. We leveraged our learnings about data frame propagation through the network from previous labs to implement the index server and client programs. This project also added another layer of complexity to the mix. In previous labs, we only managed client and server communications. This time, we had to also manage the communications between his peers. As such, to properly implement this project, We learned about various programming solutions we could use to manage multiple connections concurrently. The additional learning we had to do to overcome the challenges that arose in the development of the programs helped enrich our learning/understanding of network programming. It also provides us with an insight into the type of programming required to implement the various computer programs we use daily. In conclusion, this project was an insightful learning experience and a great conclusion to the labs. It is inspiring to see what is possible with socket programming expertise. The whole network becomes your playground.

Appendix

Index Server Code

```c
/**
 * COE 768 Final Project
 * Index Server
 *
 * Name               : Yadu Krishnan Madhu
 * Section            : 06
 * Student Number     : 500975010
 *
 * Partner Name       : Tamim Rahman (Client)
 */

#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>
#include <sys/stat.h>

#define MAX_SIZE 255

// General PDU format
struct pdu
{
    char type;
    char data[MAX_SIZE];
};

// General structure for the Send protocol. Type must be 'A' for ack, 'S' for search
// and 'R' for conttent registration
struct SPDU
{
    char type;
    char address[MAX_SIZE];
};

// General structure of registered peer
struct RegisteredPeer
{
    char type;
    char name[MAX_SIZE];
    char contentname[MAX_SIZE];
    char address[MAX_SIZE];
};
struct RegisteredPeer registry[10]; // This registry contains all the entries

// function returns whether the recieved entry exists in the registry (1 if exists, 0 otherwise).
int checkForEntry(size_t numElements, const char *peer_name, const char *content_name)
{
    int i;
    for (i = 0; i < numElements; ++i)
    {
        if (strcmp(registry[i].name, peer_name) == 0 && strcmp(registry[i].contentname, content_name) == 0)
        {
            return 1;
        }
    }
    return 0;
}

// function returns the pointer to the required entry in the register. NULL if not found (error)
const char *findAddr(size_t numElements, const char *peer_name, const char *content_name)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%s \n", registry[i].name);
        printf("%s \n", registry[i].contentname);

        if (strcmp(registry[i].name, peer_name) == 0 && strcmp(registry[i].contentname, content_name) == 0)
        {
            return registry[i].address;
        }
    }
    return NULL;
}

// function adds the entry into the registry, throws error if exceeds max number of entries (10).
void addEntry(int index, const char *s, const char *f, const char *c)
{
    // Check if the index is within bounds
    if (index >= 0 && index < 10)
    {
        strncpy(registry[index].name, s, sizeof(registry[index].name) - 1);
        strncpy(registry[index].contentname, f, sizeof(registry[index].contentname) - 1);
        strncpy(registry[index].address, c, sizeof(registry[index].address) - 1);

        // Set up string termination for peer name, content name and address.
        registry[index].name[sizeof(registry[index].name) - 1] = '\0';
        registry[index].contentname[sizeof(registry[index].contentname) - 1] = '\0';
        registry[index].address[sizeof(registry[index].address) - 1] = '\0';
    }
}

// Prints all the contents of the server
void printContentServers(size_t numElements)
{
    int i;
    for (i = 0; i < numElements; i++)
    {
        if (!checkIndex(i))
        {
            printf("Peer Name: %s \n", registry[i].name);
            printf("Content Name: %s \n", registry[i].contentname);
            printf("Content Address: %s \n\n", registry[i].address);
        }
    }
}

int checkIndex(size_t index)
{
    // Check if the peername field is an empty string or some default value
    return registry[index].name[0] == '\0';
}

int main(int argc, char *argv[])
{
    struct sockaddr_in fsin;        // the from address of a client
    struct RegisteredPeer spdu;     // Struct for send PDU
    struct pdu response;            // Struct pdu for sending back reponse
    int sock, n, addrLen;           // server socket, n to keep track, address length
    struct sockaddr_in sin;         // Incoming socket connection
    int s, type;                    // socket descriptor and socket type
    int registeredServersIndex = 0; // setting the default index location,
    char *portNum = "5000";         // Setting port number
    char buf[100];                  // input buffer to temporarily store incoming packets
    char *pts;

    switch (argc)
    {
    case 1:
        break;
    case 2:
        portNum = argv[1];
        break;
    }
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;

    // Setting up port number
    sin.sin_port = htons((__u_short)atoi(portNum));

    // Allocating socket
    s = socket(AF_INET, SOCK_DGRAM, 0);

    // Bind the socket
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    {
        fprintf(stderr, "Unable to bind socket\n");
    }

    while (1)
    {

        addrLen = sizeof(fsin);

        // reading incoming UDP connection for registering/deregistering content from peers
        if ((n = recvfrom(s, (void *)&spdu, sizeof(spdu), 0, (struct sockaddr *)&fsin, &addrLen)) < 0)
        {
            fprintf(stderr, "Error: unable to receive UDP packets\n");
        }

        // Following section checks for the incoming PDU type and executes appropriate actions based on it

        // If 'S' type then execute the search for the content within the registry
        if (spdu.type == 'R')
        {
            printf("New content Registration\n");

            // Checks for entry. If entry already exists, then error is sent back since duplicate entries are not
allowed
            if (!checkForEntry(10, spdu.name, spdu.contentname))
            {
                response.type = 'A';
                addEntry(registeredServersIndex, spdu.name, spdu.contentname, spdu.address);
                printf("Successfully registered new content and server.\n");
                printContentServers(10);
                registeredServersIndex++;
                (void)sendto(s, &response, strlen(response.data) + 1, 0, (struct sockaddr *)&fsin, sizeof(fsin));
            }
            else
            { // If no dupe is found, a ACK is sent back to indicate success.
                printf("Error: Peer name and content name already registered\n");
                response.type = 'E';
                (void)sendto(s, &response, strlen(response.data) + 1, 0, (struct sockaddr *)&fsin, sizeof(fsin));
            }

        } // If 'R' type, then a registration of the content is executed
        else if (spdu.type == 'S')
        {
            printf("Content Download request sent by content client \n");
            printf("Recieved Peer name is: %s\n", spdu.name);
            printf("Recieved Content name is %s\n", spdu.contentname);

            // Retrieves the address of the content server which contains the required content
            const char *foundAddress = findAddr(10, spdu.name, spdu.contentname);

            if (foundAddress == NULL)
            {
                printf("Error: Content entry not found\n");
                printf("List of available content: \n");
                printContentServers(10);
            }
            else
            {
                response.type = 'S';
                strncpy(response.data, foundAddress, MAX_SIZE);
                printf("Success: Content Found\n");
            }
            // Sends the reponse
            (void)sendto(s, &response, strlen(response.data) + 1, 0, (struct sockaddr *)&fsin, sizeof(fsin));
        }
    }
}
```

Client Code:

```
/* p2p_client.c - main

Name: Tamim Rahman (Client)
ID: 500967494
Section: 03
Partner: Yadu Krishnan Madhu (Server)

*/
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>

#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <netdb.h>
#include <errno.h>

#define BUFSIZE 64
# define    h_addr  h_addr_list[0] /* Address, for backward compatibility.*/


struct pdu
{
    char type;
    char data[255];
};

struct DPDU
{
    char type;
    char contentName[255];
};

struct APDU
{
    char type;
    char peerName[255];
    char contentName[255];
};

struct RPDU
{
    char type;
    char peerName[255];
    char contentName[255];
    char address[255];
};

char *removeSpace(char *str)
{
    char *end;
    while (isspace((unsigned char)*str))
    {
        str += 1;
    }
    if (*str == 0)
    {
        return str;
    }
    end = str + strlen(str) - 1;
    while ((isspace((unsigned char)*end) && (end > str)))
    {
        end -= 1;
    }
    end[1] = '\0';
    return str;
}

int main(int argc, char *argv[])
{
    int s, n, type;
    char *host = "placeholder";
    char *port = "5000";

    port = argv[2];
    host = argv[1];

    struct hostent *phe;
    struct sockaddr_in sin;

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

    sin.sin_port = htons((__u_short)atoi(port));
    sin.sin_addr.s_addr = inet_addr(host);
    phe = gethostbyname(host);

    if (phe)
    {
```

```
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
}
else if (INADDR_NONE == sin.sin_addr.s_addr)
{
    fprintf(stderr, "Error: Connection issue \n");
    exit(1);
}

s = socket(PF_INET, SOCK_DGRAM, 0);
if (s < 0)
{
    fprintf(stderr, "Error: Unable to create the socket\n");
    exit(1);
}

if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
{
    fprintf(stderr, "Error: Unable to connect on %s %s \n", host, port);
    exit(1);
}
struct RPDU client_content_pdu;
struct APDU search_pdu;
struct pdu cpdu;
struct pdu res;
struct DPDU dpdu;
struct RPDU content_pdu;
struct pdu cpdu_res;

while (1)
{
    int user_choice;
    printf("---------P2P Content Downloader----------\n");
    printf("What would you like to do?: \n");
    printf("1. Host Content \n");
    printf("2. Download Content \n");
    scanf("%d", &user_choice);
    if (user_choice == 1)
    {
        struct sockaddr_in content_addr;
        content_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        content_addr.sin_port = htons(0);
        content_addr.sin_family = AF_INET;
        int stcp;
        int addr_len;
        stcp = socket(AF_INET, SOCK_STREAM, 0);

        if (bind(stcp, (struct sockaddr *)&content_addr, sizeof(content_addr)) < 0)
        {
            fprintf(stderr, "Error: Socket Bind issue");
        }
        char con_host_ip[INET_ADDRSTRLEN];
        char full_address[255];
        addr_len = sizeof(struct sockaddr_in);
        int check = getsockname(stcp, (struct sockaddr *)&content_addr, &addr_len);
        char ip_address[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(content_addr.sin_addr), ip_address, INET_ADDRSTRLEN);

        content_pdu.type = 'R';
        printf("Please enter your ip address: \n");
        scanf("%s", con_host_ip);
        printf("Please select a name for your peer: \n");
        scanf("%s", content_pdu.peerName);
        printf("Please enter the name of the content you would like to host: \n");
        scanf("%s", content_pdu.contentName);
```

```
snprintf(full_address, sizeof(full_address), "%s:%d", con_host_ip, ntohs(content_addr.sin_port));
strcpy(content_pdu.address, full_address);

write(s, &content_pdu, 765);
n = read(s, &res, sizeof(res));
if (res.type == 'A')
{
    printf("Index Server successfully registered your content!\n");
    if (listen(stcp, 5) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    printf("Receiving content download requests on %d\n", ntohs(content_addr.sin_port));
    fd_set rfds, afds;
    FD_ZERO(&afds);
    FD_SET(stcp, &afds);
    FD_SET(0, &afds);
    int new_sd;
    char buf[255];
    struct sockaddr_in client;

    while (1)
    {
        memcpy(&rfds, &afds, sizeof(rfds));

        if (select(FD_SETSIZE, &rfds, NULL, NULL, NULL) < 0)
        {
            perror("select");
            exit(EXIT_FAILURE);
        }

        if (FD_ISSET(0, &rfds))
        {
            n = read(0, buf, BUFSIZE);
            printf("buf here is %s\n", buf);
        }

        if (FD_ISSET(stcp, &rfds))
        {
            new_sd = accept(stcp, (struct sockaddr *)&client, &addr_len);
            if (new_sd != -1)
            {
                printf("Content Registration\n");
                char sbuf[255];
                int n;
                struct DPDU serverres;
                n = read(new_sd, &serverres, 255);

                if (serverres.type == 'D')
                {
                    cpdu.type = 'C';
                    char *rem;
                    rem = removeSpace(serverres.contentName);

                    FILE *fp = fopen(rem, "r");
                    if (fp != NULL)
                    {
                        size_t bytesRead;
                        while ((n = fread(sbuf, 1, 255, fp)) > 0)
                        {
                            strcpy(cpdu.data, sbuf);
                            write(new_sd, &cpdu, 255);
                        }
                        fclose(fp);
```

```
                            close(stcp);
                        }
                        else
                        {
                            printf("Error: File not found \n");
                            write(stcp, "NO FILE", n);
                            close(stcp);
                            return 1;
                        }
                        return 0;
                    }
                }
            }
        }
    }
    else if (res.type = 'E')
    {
        printf("Peer name already exists.\n");
        printf("Please choose another peer name.\n");
    }
}

else if (user_choice == 2)
{
    search_pdu.type = 'S';
    printf("Please enter the name of the content you would like to downlaod: \n");
    scanf("%s", search_pdu.contentName);
    printf("Please enter the name of the peer you would like to download from: \n");
    scanf("%s", search_pdu.peerName);

    write(s, &search_pdu, 510);
    n = read(s, &res, sizeof(res));
    if (n < 0)
    {
        fprintf(stderr, "Error: Issue with reading response\n");
    }

    if (res.type == 'S')
    {
        printf("Index Server: Content has been found!\n");
        printf("Address for Content: %s\n", res.data);

        char str[128];
        char *ptr;
        strcpy(str, res.data);
        strtok_r(str, ":", &ptr);
        printf("'%s'  '%s'\n", str, ptr);

        int sd;
        struct sockaddr_in server;
        struct hostent *hp;
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
            fprintf(stderr, "Error: Unable to form socket\n");
            exit(1);
        }

        bzero((char *)&server, sizeof(struct sockaddr_in));
        server.sin_family = AF_INET;
        server.sin_port = htons(atoi(ptr));
        if (hp = gethostbyname(str))
            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);
        }
        if (connect(sd, (struct sockaddr *)&server, sizeof(server)) == -1)
        {
            fprintf(stderr, "Can't connect \n");
            exit(1);
        }

        char cbuf[255];
        printf("Connected to content server!\n");

        dpdu.type = 'D';
        strcpy(dpdu.contentName, search_pdu.contentName);
        size_t contentNameLength = strlen(search_pdu.contentName);

        printf("File name sent to content server: %s\n", search_pdu.contentName);
        size_t bytes_written = write(sd, &dpdu, 255);

        if (bytes_written < 0)
        {
            perror("Error writing to socket");
            // Handle the error, e.g., return or exit
        }
        printf("Bytes written from client: %d\n", bytes_written);
        printf("Preparing to send you content...\n");

        char *rem = removeSpace(search_pdu.contentName);
        char rbuf[255];

        FILE *fp = fopen(rem, "w");

        if (fp == NULL)
        {
            perror("Error opening file for writing.");
            close(sd);
            exit(1);
        }

        while ((n = read(sd, &cpdu_res, 255)) > 0)
        {
            if (cpdu_res.type == 'C')
            {
                fprintf(fp, cpdu_res.data);
            }
        }

        fclose(fp);
        printf("Success: File has been downloaded!\n");
        close(sd);

        char con_host_ip[INET_ADDRSTRLEN];

        printf("Transforming into a content serving peer...\n");
        int stcp;
        struct sockaddr_in content_addr;
        int addr_len;
        stcp = socket(AF_INET, SOCK_STREAM, 0);
        content_addr.sin_family = AF_INET;
        content_addr.sin_port = htons(0);
        content_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(stcp, (struct sockaddr *)&content_addr, sizeof(content_addr)) < 0)
        {
            perror("Error: Socket binding error");
            exit(EXIT_FAILURE);
        }
```

```
addr_len = sizeof(struct sockaddr_in);
int check = getsockname(stcp, (struct sockaddr *)&content_addr, &addr_len);

char ip_address[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &(content_addr.sin_addr), ip_address, INET_ADDRSTRLEN);

char full_address[255];

printf("Please enter your ip address: \n");
scanf("%s", con_host_ip);
printf("Enter your peer name: \n");
scanf("%s", client_content_pdu.peerName);

client_content_pdu.type = 'R';
strcpy(client_content_pdu.contentName, search_pdu.contentName);
snprintf(full_address, sizeof(full_address), "%s:%d", con_host_ip, ntohs(content_addr.sin_port));

printf("Peer address and port: %s\n", full_address);
strcpy(client_content_pdu.address, full_address);

write(s, &client_content_pdu, 765);

n = read(s, &res, sizeof(res));
if (n < 0)
{
    fprintf(stderr, "Read failed\n");
}

if (res.type == 'A')
{
    printf("Index Server successfully registered your content!\n");
    if (listen(stcp, 5) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Content is hosted on port: %d \n", ntohs(content_addr.sin_port));
}

fd_set rfds, afds;
FD_ZERO(&afds);
FD_SET(stcp, &afds);
FD_SET(0, &afds);

int new_sd;
char buf[255];
struct sockaddr_in client;

while (1)
{
    memcpy(&rfds, &afds, sizeof(rfds));

    if (select(FD_SETSIZE, &rfds, NULL, NULL, NULL) < 0)
    {
        perror("select");
        exit(EXIT_FAILURE);
    }
    if (FD_ISSET(0, &rfds))
    {
        n = read(0, buf, BUFSIZE);
        printf("buf here is %s\n", buf);
    }
```

```
                    if (FD_ISSET(stcp, &rfds))
                    {
                        new_sd = accept(stcp, (struct sockaddr *)&client, &addr_len);
                        if (new_sd != -1)
                        {
                            printf("Received content download request!\n");

                            char sbuf[255];
                            int n;
                            struct DPDU serverres;
                            n = read(new_sd, &serverres, 255);

                            if (serverres.type == 'D')
                            {
                                cpdu.type = 'C';
                                char *rem;
                                rem = removeSpace(serverres.contentName);
                                FILE *fp = fopen(rem, "r");
                                if (fp == NULL)
                                {
                                    printf("Error: File not found \n");
                                    write(stcp, "NO FILE", n);
                                    close(stcp);
                                    return 1;
                                }
                                else
                                {
                                    size_t bytesRead;
                                    while ((n = fread(sbuf, 1, 255, fp)) > 0)
                                    {
                                        strcpy(cpdu.data, sbuf);
                                        write(new_sd, &cpdu, 255);
                                    }
                                    fclose(fp);
                                    close(stcp);
                                }
                                return 0;
                            }
                        }
                    }
                }
                close(sd);
                return 0;
            }
        }
        else if (res.type = 'E')
        {
            printf("Error: Content not found!\n");
        }
    }
}
```