RYERSON UNIVERSITY

Faculty of Engineering, Architecture and Science

Department of Electrical and Computer Engineering

Course Number	768
Course Title	Computer Networks
Semester/Year	F2022

Instructor	Dr. Truman Yang

Project No.	1

Project Title	P2P Application

Submission Date	December 2nd, 2022
Due Date	December 2nd, 2022

Student Name	Student ID	Signature*
Abdulrehman Khan	500968727	A.K.
Hamza Iqbal	500973673	H.I.
Fatima Rahman	500892014	F.R

**By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work.*

Table of Contents

Background

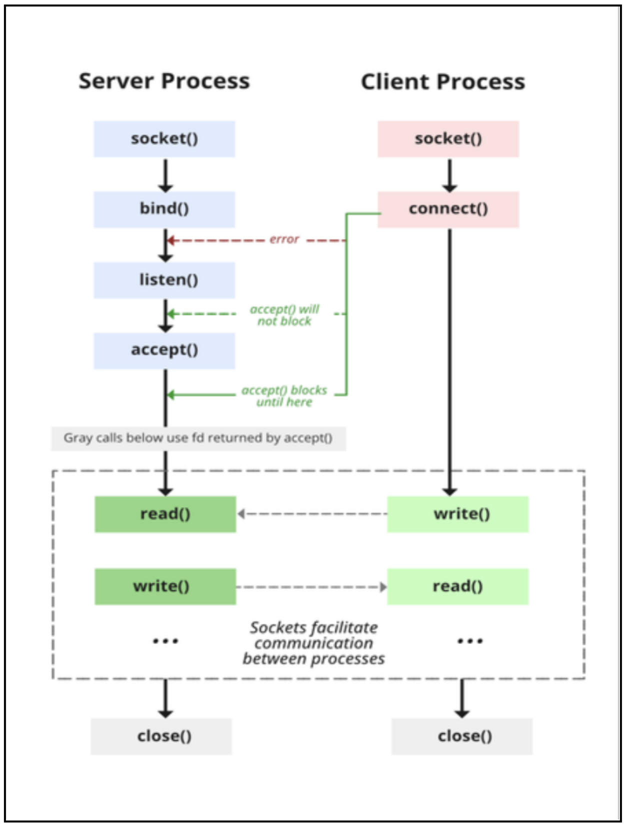
Socket APIs are used in socket programming to establish communication links on a network between local and remote processes (*Socket Programming*, n.d.). Specifically, socket programming uses two nodes on a network to communicate with each other where each socket has a specific role (GeeksforGeeks, 2022). Typically, one socket (node) *listens* on a port at an IP, where both port and IP are unique. Then, the second socket *reaches* out to the "first" socket to establish a connection (GeeksforGeeks, 2022). This concept is broken up into the server endpoint and the client endpoint.

The server process is as follows:

1. Create a socket(): This function initializes the appropriate socket and its protocols, such as, domain, type, SOCK_STREAM, SOCK_DGRAM, protocol
2. Bind() the socket: The function binds the socket to the address and port number specified in addr(custom data structure).
3. Listen(): At this point the server enters into a standby mode where it awaits for the client to establish a connection. The function defines the maximum length of a pending connection for the socket. Otherwise, the function provides an error if the queue is full when the connection request arrives
4. Accept(): This function utilizes the first connection request in the queue and establishes a connection between the client and the server.

The client process:

1. Create a socket()
2. Request to connect() to the server: This function is received after the bind() function in the server process. This call is accepted by the server and thus results in an established connection.



;
Figure 1: State diagram for the server and client model of a Socket (GeeksforGeeks, 2022)

Moreover, socket programming cannot be successful without the help of TCP and UDP protocols. Transmission Control Protocol (TCP) is a connection-oriented data that can be sent in two directions: To/From Client/Server. The TCP is preferred for security issues as it has a built in system that checks for errors. TCP follows a queue data structure, so each data packet will be delivered in the order it was sent, making it the perfect protocol for transferring various types of data such as digital images, text files, and web pages.

In contrast, User Datagram Protocol (UDP) is a simpler, connectionless Internet protocol where error-checking and recovery services are not required. With UDP, the data is continuously flowing to the recipient even if they do not receive it as the process ignores any error checking such as opening, maintaining, or terminating a connection (*TCP Vs UDP*, n.d.).

UDP is preferred for real-time communications like broadcast and multi-task network transmission (*TCP Vs UDP*, n.d.).

.

Introduction

The objective of this project is to develop a Peer-to-Peer (P2P) application that consists of an index server and a number of peers. The application should permit the peers to exchange content through the support of the index server, as well as, download content through a content server. The communication between the index server and a peer is based on UDP while the content download is based on TCP. The following figure illustrates the P2P mechanism. This application is developed using the knowledge of Labs 1, 2 , 3 and 4 and the Comlab Virtual Linux. (COE 768, 2022).

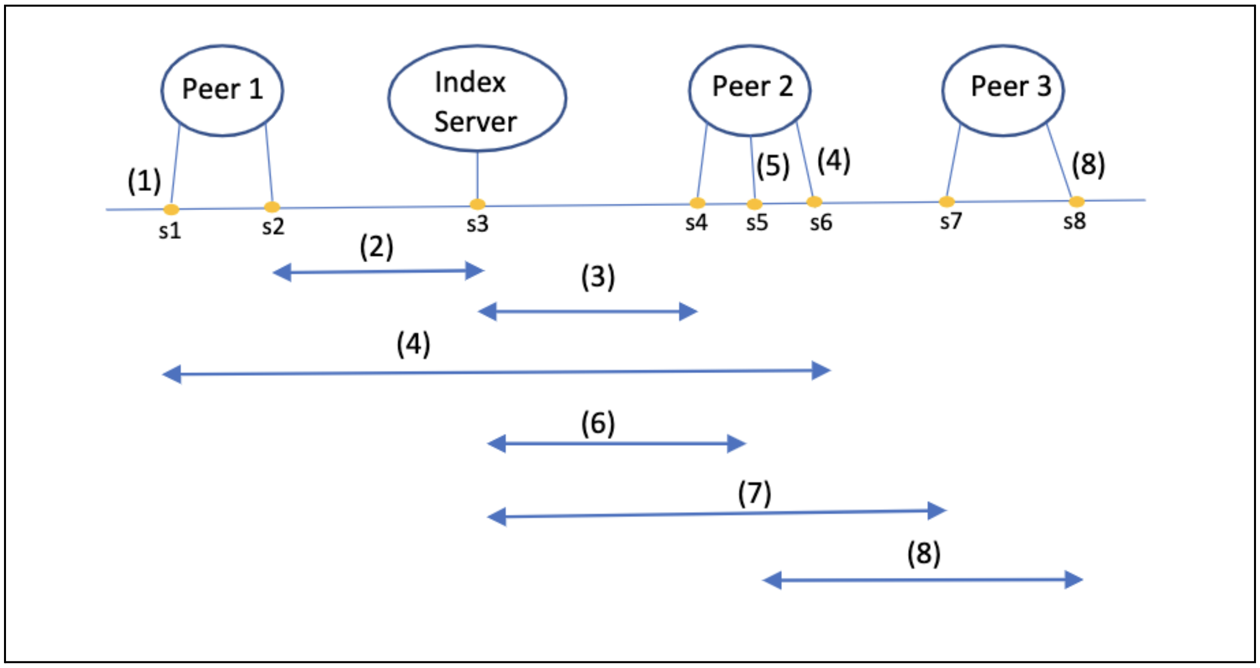


Figure 2: Diagram of the Peer to Peer mechanism

The figure demonstrates eight different sockets, where sockets 2, 3 4 and 7 are UDP sockets while sockets 1 and 4 are TCP sockets. As defined previously, the UDP sockets are simpler and connectionless, so these sockets focus on communicating with the index server without worrying about any external factors such as losing information.

Design Specifications

Programming Requirements

This project defines three key characters, Peer 1, Peer 2 and Peer 3. Peer 1 has the role of creating content to be registered with the index server. Next, after content registration, Peer 2 makes a connection with the index server so that it can search for the content that has been registered by Peer 1. Peer 2 searches the index server so that it can download this content. Once the download is complete, Peer 2 registers itself on the same content server. Then, Peer 3 repeats the same tasks as Peer 2, where it searches for the content from the respective index server, however, instead of receiving the content from Peer 1, it receives it from Peer 2. Once the server responds with the appropriate content, Peer 3 proceeds to download the information from Peer 2.

<u>Protocol Data Unit Format</u>

The protocol data unit (PDU) is a data structure type used to communicate with networking protocols. When working with a multilayer protocol stack, like the TCP/IP networking suite, use of the correct PDU is important when discussing protocol interactions. The PDU has the following format:

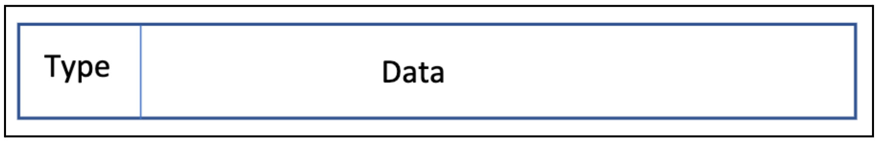


Figure 3: Diagram of the PDU

There are eight total PDU types, where the Type field has the size of one byte and specifies the PDU type.

<u>Implementation of Protocol</u>

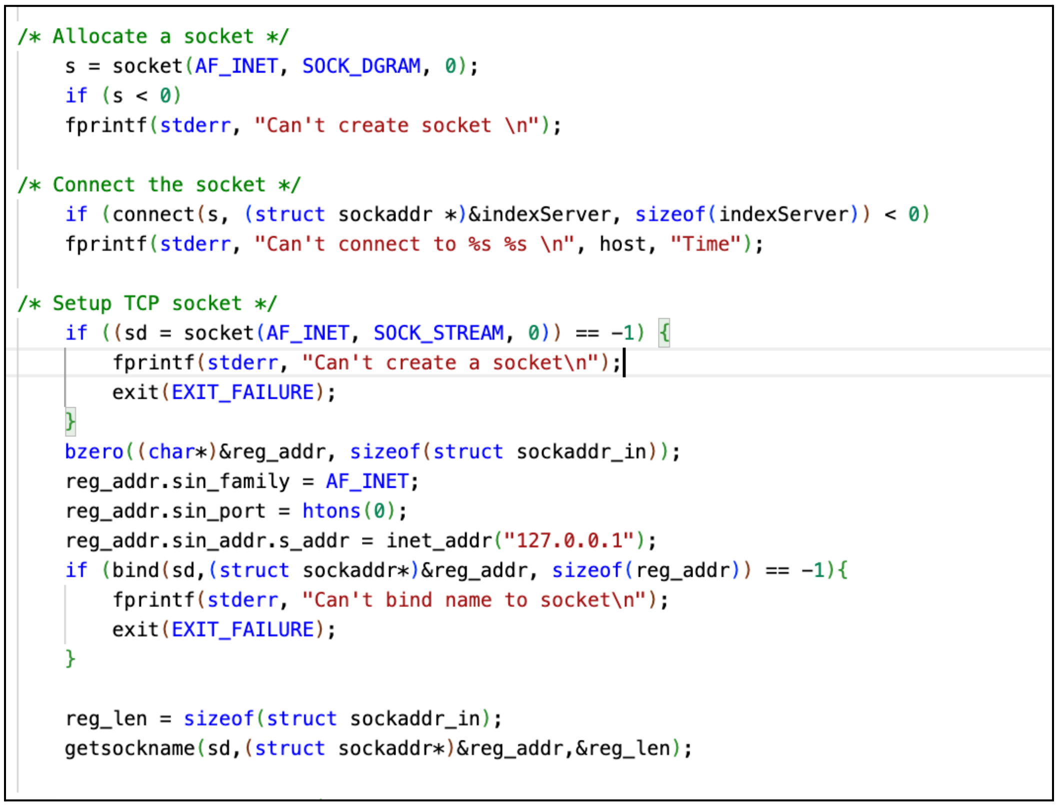
The PDU plays an important role in the development of Peer to Peer communication. The following table summarizes the various types of PDU used in this application:

PDU type	Function	Direction
R	Content Registration	Peer to Index Server
D	Content Download Request	Content Client to Content Server
S	Search for content and the associated content server	Between Peer and Index Server
T	Content De-Registration	Peer to Index Server
C	Content Data	Content Server to Content Client
O	List of On-Line Registered Content	Between Peer and Index Server
A	Acknowledgement	Index Server to Peer
E	Error	Between Peers or between Peer and Index Server

Table 1: Table demonstrating the various PDU types and their functions.

<u>Initialization of the Sockets</u>

Before implementing the Peer to Peer communication, a socket environment must be set up and established. The following code demonstrates how a socket is first allocated, then connected before setting up a TCP socket:



```
/* 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 *)&indexServer, sizeof(indexServer)) < 0)
    fprintf(stderr, "Can't connect to %s %s \n", host, "Time");

/* Setup TCP socket */
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        fprintf(stderr, "Can't create a socket\n");
        exit(EXIT_FAILURE);
    }
    bzero((char*)&reg_addr, sizeof(struct sockaddr_in));
    reg_addr.sin_family = AF_INET;
    reg_addr.sin_port = htons(0);
    reg_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    if (bind(sd,(struct sockaddr*)&reg_addr, sizeof(reg_addr)) == -1){
        fprintf(stderr, "Can't bind name to socket\n");
        exit(EXIT_FAILURE);
    }

    reg_len = sizeof(struct sockaddr_in);
    getsockname(sd,(struct sockaddr*)&reg_addr,&reg_len);
```

Program Breakdown

Purpose of Client Program

The communication between a peer and the index server is based on UDP. The client program is focused on delivering the users input to the index server. Specifically, the client will prompt the user with a set of questions where they are to respond with the appropriate commands provided by the Protocol Data Unit Format (PDU).

A peer can act as either a content server of a given piece of content, or a content client that has its own unique content to be delivered upon request.

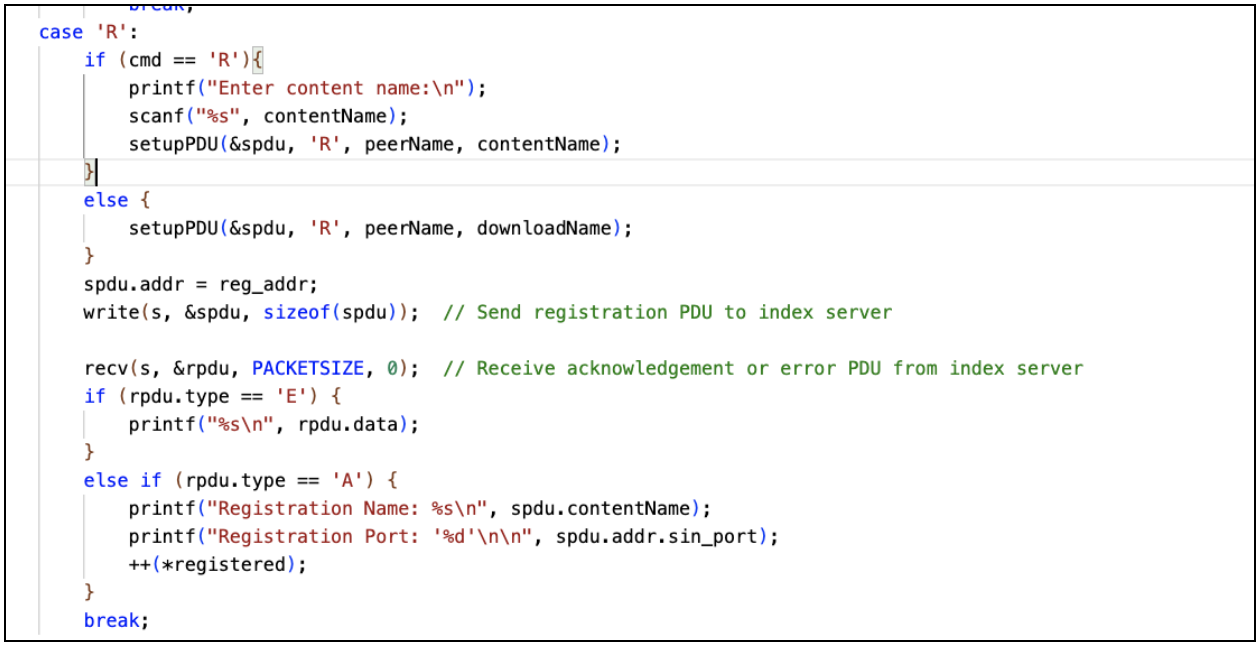
Purpose of Server Program

The server program follows the TCP and UDP format and receives information delivered by the user input and conducts the appropriate processes. In particular, the server program will create both an index server. The index server acts as a platform for the Peers 1, 2, and 3 to communicate through. The index server provides the Peers a way to access the content server. The content server is responsible for receiving the content registered by a given Peer and delivering the content to the requested Peer, where each Peer is uniquely assigned. Once the content is downloaded from the index server it then becomes the content server.

Content Registration

The first step of the application is to have a Peer register content to the index server. In order for this to occur, a switch case data structure is implemented to accommodate the command R from the PDU types. This switch case structure also accommodates the other commands provided by the PDU which is utilized in other cases.

The beginning of the program prompts the user to submit a command. Given that Peer 1 is to register a content, the command is then "R". Following the command "R", the switch case data structure prompts the client side to provide the name of the content they want to be registered:

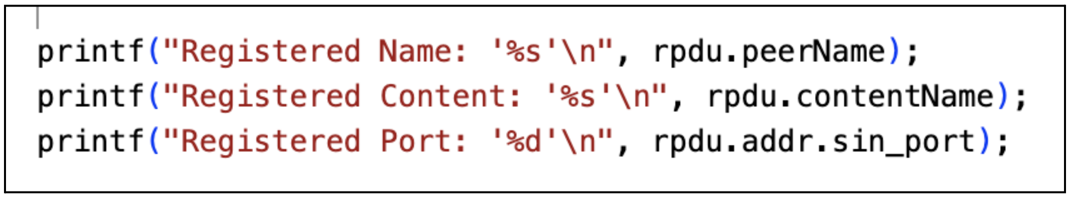


```
case 'R':
    if (cmd == 'R'){
        printf("Enter content name:\n");
        scanf("%s", contentName);
        setupPDU(&spdu, 'R', peerName, contentName);
    }
    else {
        setupPDU(&spdu, 'R', peerName, downloadName);
    }
    spdu.addr = reg_addr;
    write(s, &spdu, sizeof(spdu));  // Send registration PDU to index server

    recv(s, &rpdu, PACKETSIZE, 0);  // Receive acknowledgement or error PDU from index server
    if (rpdu.type == 'E') {
        printf("%s\n", rpdu.data);
    }
    else if (rpdu.type == 'A') {
        printf("Registration Name: %s\n", spdu.contentName);
        printf("Registration Port: '%d'\n\n", spdu.addr.sin_port);
        ++(*registered);
    }
    break;
```

Let the content Peer 1 intends to register be a text file labeled as "hi.txt".

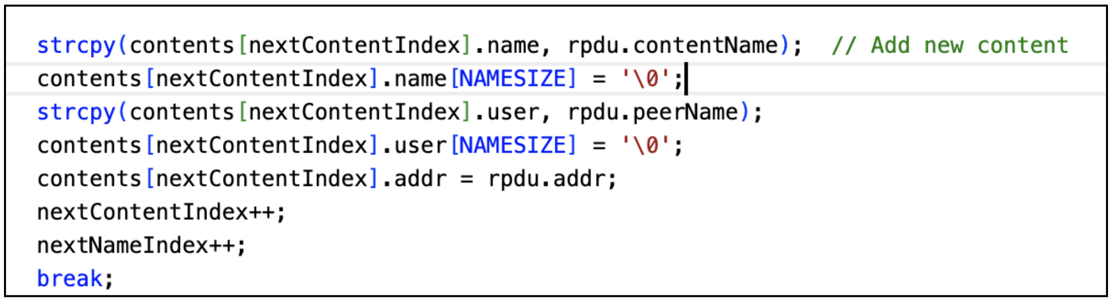
Now, on the server side, the program acknowledges the client side. This communication is based on UDP. The server program also uses a switch case data structure. When the command is "R", the server first checks if the Peer is unique. If it is not, the server sends an error message that the provided peer name already exists.

Otherwise, the server pings the client with information of the content registration:



```
printf("Registered Name: '%s'\n", rpdu.peerName);
printf("Registered Content: '%s'\n", rpdu.contentName);
printf("Registered Port: '%d'\n", rpdu.addr.sin_port);
```

And then simultaneously adds the new content to the server in an array:

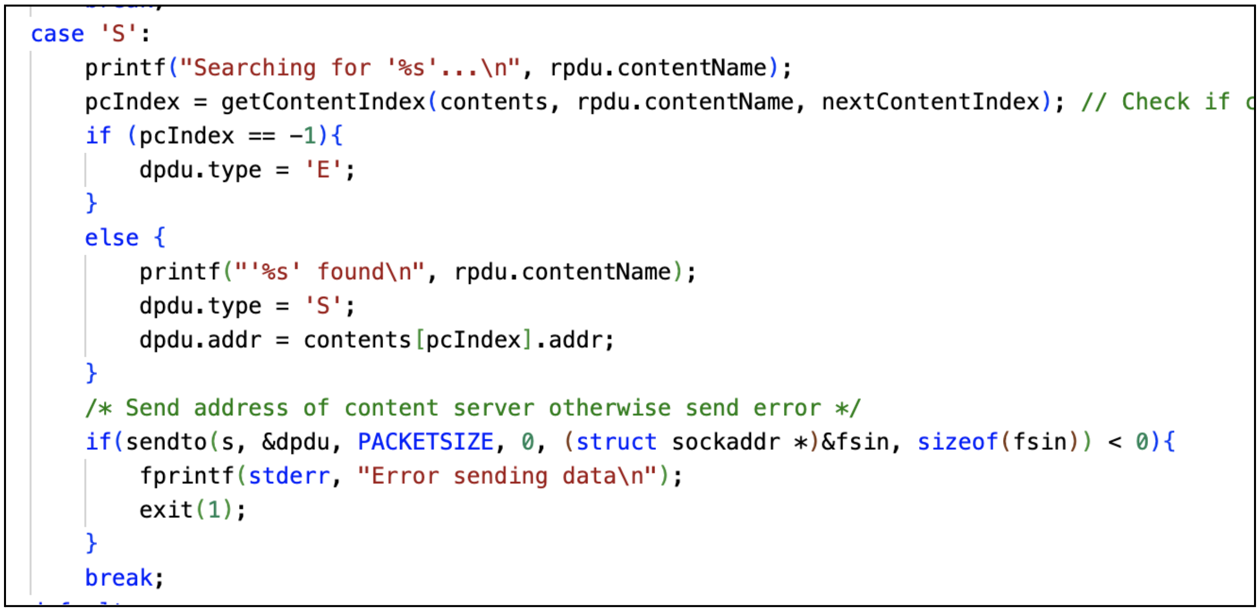


```
strcpy(contents[nextContentIndex].name, rpdu.contentName);  // Add new content
contents[nextContentIndex].name[NAMESIZE] = '\0';
strcpy(contents[nextContentIndex].user, rpdu.peerName);
contents[nextContentIndex].user[NAMESIZE] = '\0';
contents[nextContentIndex].addr = rpdu.addr;
nextContentIndex++;
nextNameIndex++;
break;
```

This process concludes content registration from Peer 1 to the index server.

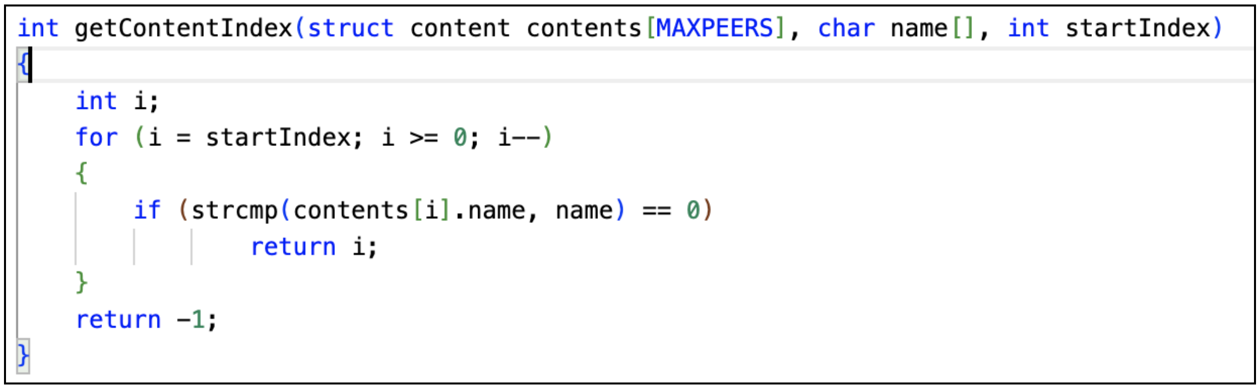
Content Search

After content registration, another Peer, Peer 2, proceeds to search for the content in the index server. In order to search for the content, Peer 2 on the client side provides the command “S”, where, on the server side it calls a function:



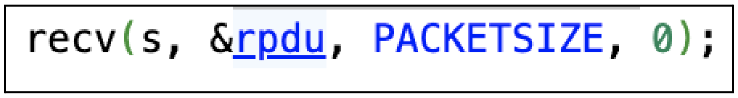
```
case 'S':
    printf("Searching for '%s'...\n", rpdu.contentName);
    pcIndex = getContentIndex(contents, rpdu.contentName, nextContentIndex); // Check if c
    if (pcIndex == -1){
        dpdu.type = 'E';
    }
    else {
        printf("'%s' found\n", rpdu.contentName);
        dpdu.type = 'S';
        dpdu.addr = contents[pcIndex].addr;
    }
    /* Send address of content server otherwise send error */
    if(sendto(s, &dpdu, PACKETSIZE, 0, (struct sockaddr *)&fsin, sizeof(fsin)) < 0){
        fprintf(stderr, "Error sending data\n");
        exit(1);
    }
    break;
```

The server calls the function getContentIndex() which calls for the content inside of the index server. It searches for the name of the file provided by the client through looping the array:



```
int getContentIndex(struct content contents[MAXPEERS], char name[], int startIndex)
{
    int i;
    for (i = startIndex; i >= 0; i--)
    {
        if (strcmp(contents[i].name, name) == 0)
                return i;
    }
    return -1;
}
```

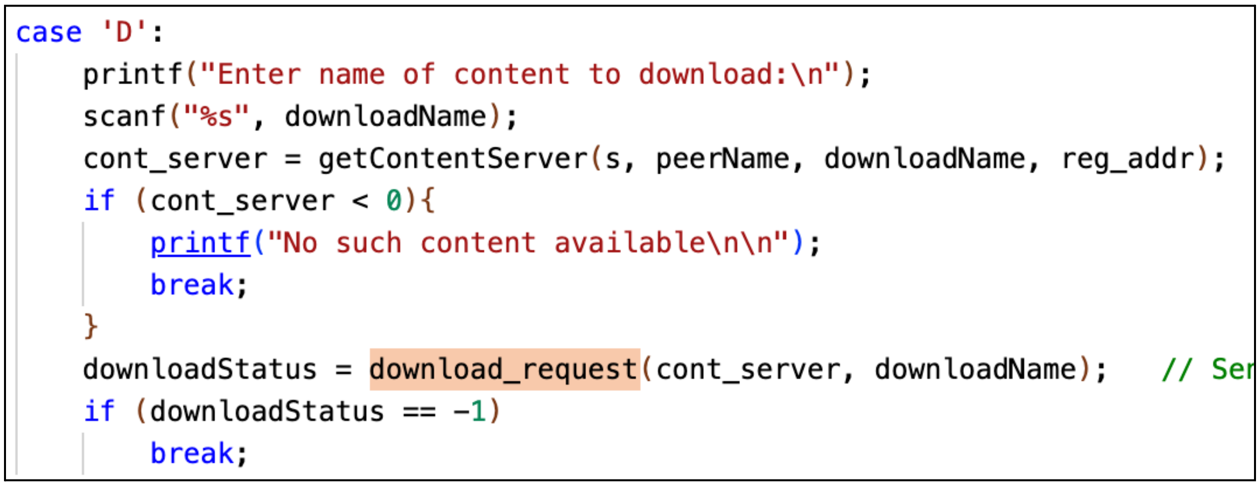
The ‘S’ command gets the information of the content server from the index server. It receives the the content server address and port from the index server:



```
recv(s, &rpdu, PACKETSIZE, 0);
```

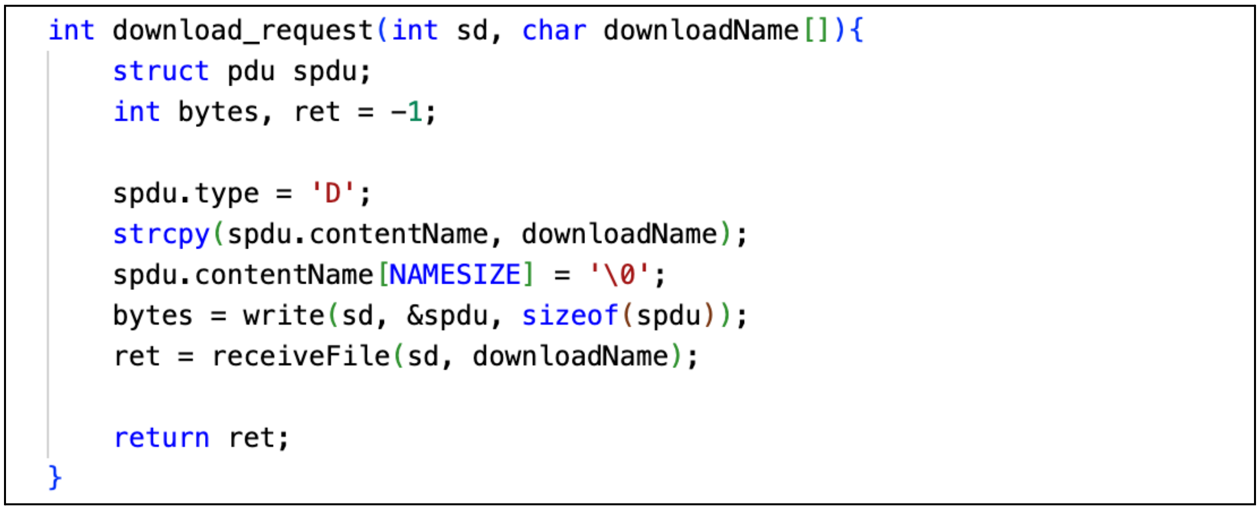
<u>Content Download</u>

Subsequently, Peer 2 can use this information to download the requested content from the content server. This is done by inputting the command “D”:



```
case 'D':
    printf("Enter name of content to download:\n");
    scanf("%s", downloadName);
    cont_server = getContentServer(s, peerName, downloadName, reg_addr);
    if (cont_server < 0){
        printf("No such content available\n\n");
        break;
    }
    downloadStatus = download_request(cont_server, downloadName);   // Ser
    if (downloadStatus == -1)
        break;
```

When command “D” is prompted, the client program calls the function download_request() which sends a download request to the content server to retrieve the file “hi.txt” through the receiveFile() function:

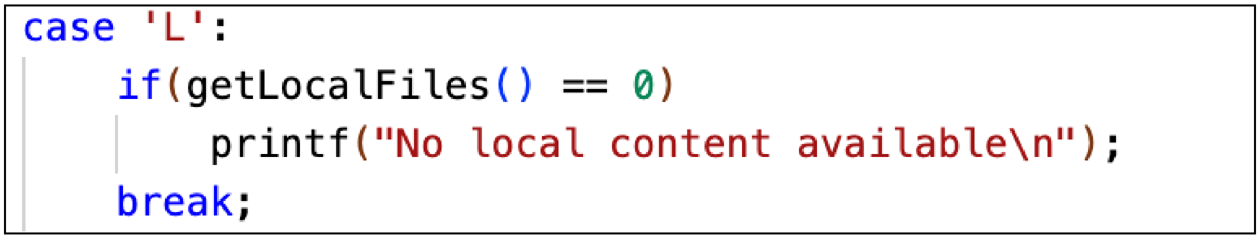


```
int download_request(int sd, char downloadName[]){
    struct pdu spdu;
    int bytes, ret = -1;

    spdu.type = 'D';
    strcpy(spdu.contentName, downloadName);
    spdu.contentName[NAMESIZE] = '\0';
    bytes = write(sd, &spdu, sizeof(spdu));
    ret = receiveFile(sd, downloadName);

    return ret;
}
```

Moreover, Peer 3 now begins the same process. It requests the same content (“hi.txt”) to be downloaded from the content server, which is now Peer 2. As Peer 2 registers the content to the index server, it becomes the server of the content itself. Thus, to distribute the load evenly of the TCP pockets, the index server chooses a content server that is least used, in this case it is Peer 2. This occurs after Peer 3 requests a search for the content.

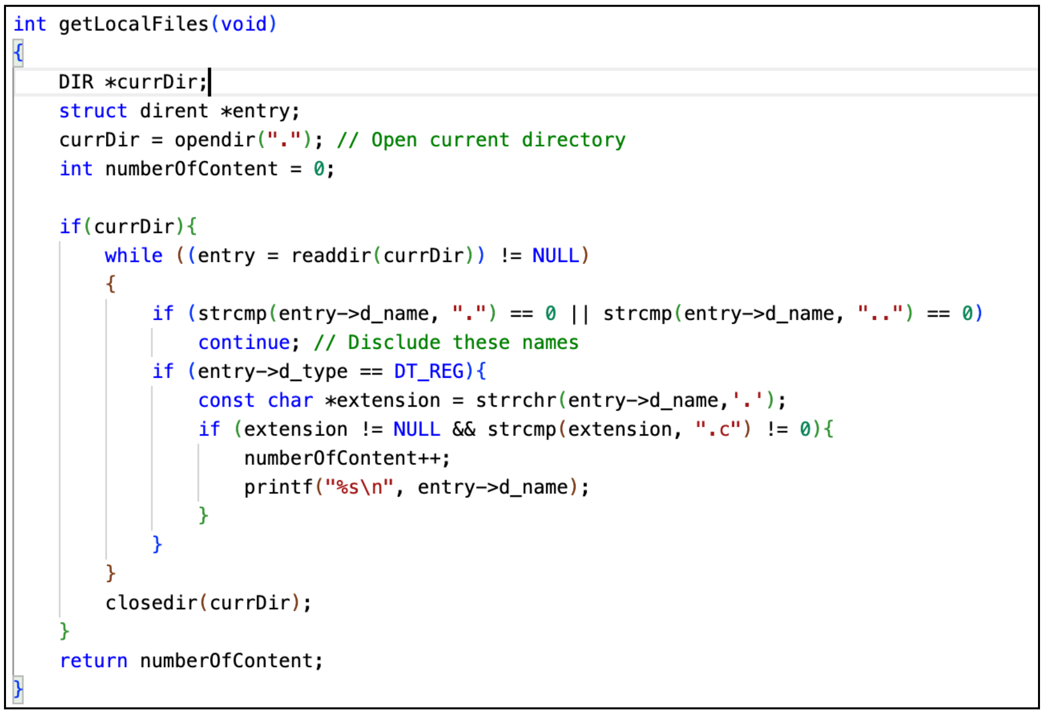
<u>Content Listing</u>

The command type "L" provided by the PDU allows a Peer to request a list of registered content from the index server.



```c
case 'L':
    if(getLocalFiles() == 0)
        printf("No local content available\n");
    break;
```

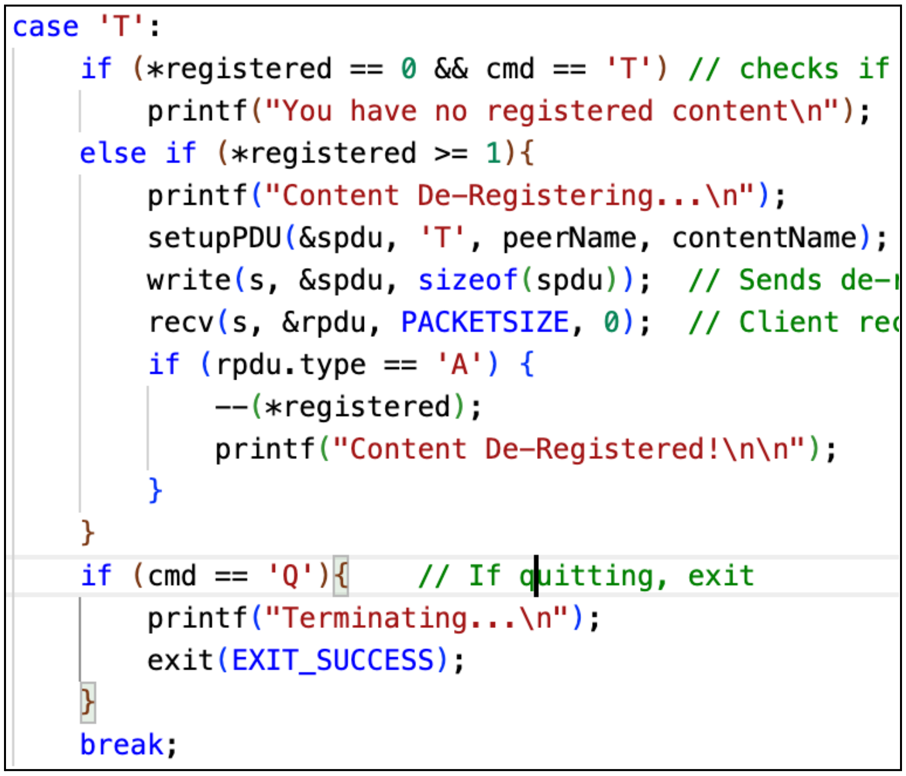
This case calls the function getLocalFiles() where it opens the current directory to check if the files exist in the server:



```c
int getLocalFiles(void)
{
    DIR *currDir;
    struct dirent *entry;
    currDir = opendir("."); // Open current directory
    int numberOfContent = 0;

    if(currDir){
        while ((entry = readdir(currDir)) != NULL)
        {
            if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
                continue; // Disclude these names
            if (entry->d_type == DT_REG){
                const char *extension = strrchr(entry->d_name,'.');
                if (extension != NULL && strcmp(extension, ".c") != 0){
                    numberOfContent++;
                    printf("%s\n", entry->d_name);
                }
            }
        }
        closedir(currDir);
    }
    return numberOfContent;
}
```

<u>Content De-Registration</u>

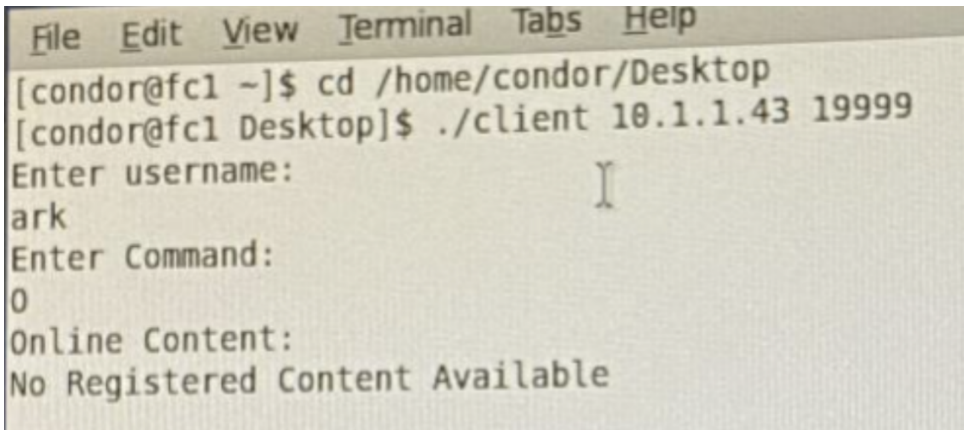
A Peer can deregister content from the server with the command "T". This case tests to see if the content exists first, and if it does not then it sends out an error message. Otherwise, it continues to send a deregistration PDU message to the index server and simultaneously awaits for an acknowledgement packet confirming the deregistration.



```
case 'T':
    if (*registered == 0 && cmd == 'T') // checks if
        printf("You have no registered content\n");
    else if (*registered >= 1){
        printf("Content De-Registering...\n");
        setupPDU(&spdu, 'T', peerName, contentName);
        write(s, &spdu, sizeof(spdu));  // Sends de-
        recv(s, &rpdu, PACKETSIZE, 0);  // Client rec
        if (rpdu.type == 'A') {
            --(*registered);
            printf("Content De-Registered!\n\n");
        }
    }
    if (cmd == 'Q'){     // If quitting, exit
        printf("Terminating...\n");
        exit(EXIT_SUCCESS);
    }
    break;
```

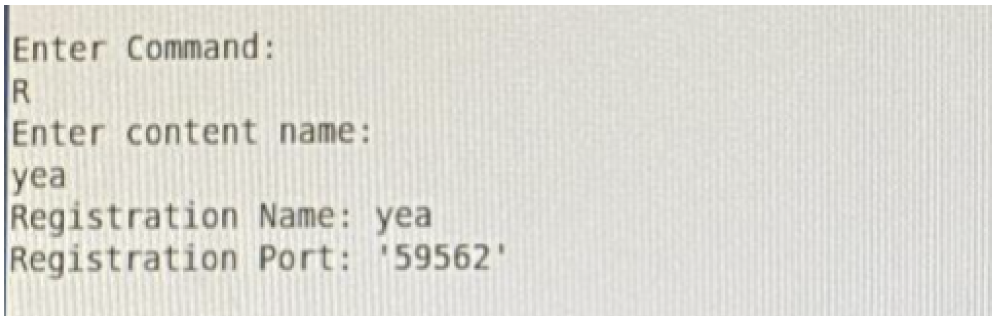
Observations and Analysis

Upon initializing the client and server, the client side prompts Peer 1 to enter in a username. Then, Peer 1 enters the command "O", a PDU type that represents a "list of online registered content" as shown in Table 1. The command results in "No Registered Content Available" which is accurate as no content has been registered by a peer yet:



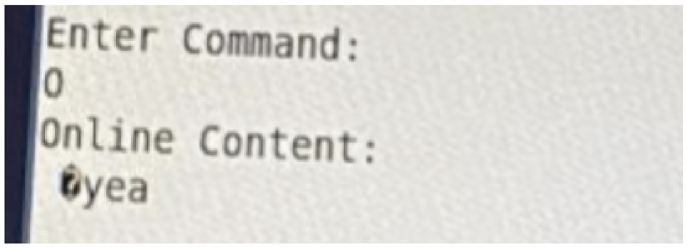
```
[condor@fc1 ~]$ cd /home/condor/Desktop
[condor@fc1 Desktop]$ ./client 10.1.1.43 19999
Enter username:
ark
Enter Command:
O
Online Content:
No Registered Content Available
```

Furthermore, Peer 1 inputs the command "R" to initialize registration. The next prompt is to enter a unique content name, which is "yea". The server then acknowledges the registration by submitting back the "Registration Name" (of the content) and its respective "Registration Port":



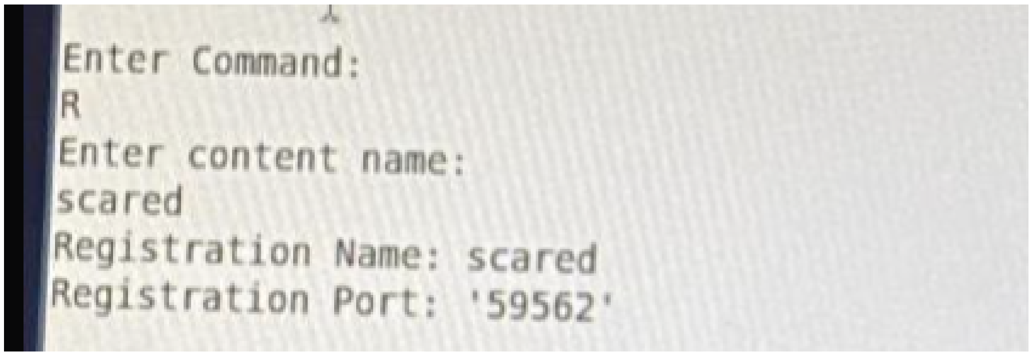
```
Enter Command:
R
Enter content name:
yea
Registration Name: yea
Registration Port: '59562'
```

By using the command "O" again, the online-list is updated and shows the most recently registered content.



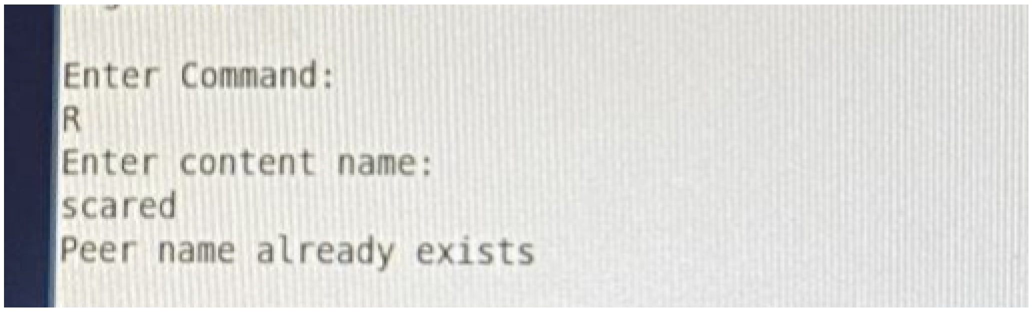
```
Enter Command:
O
Online Content:
 yea
```

This confirms the Content Registration code works successfully for Peer 1 to Index (Content) Server.



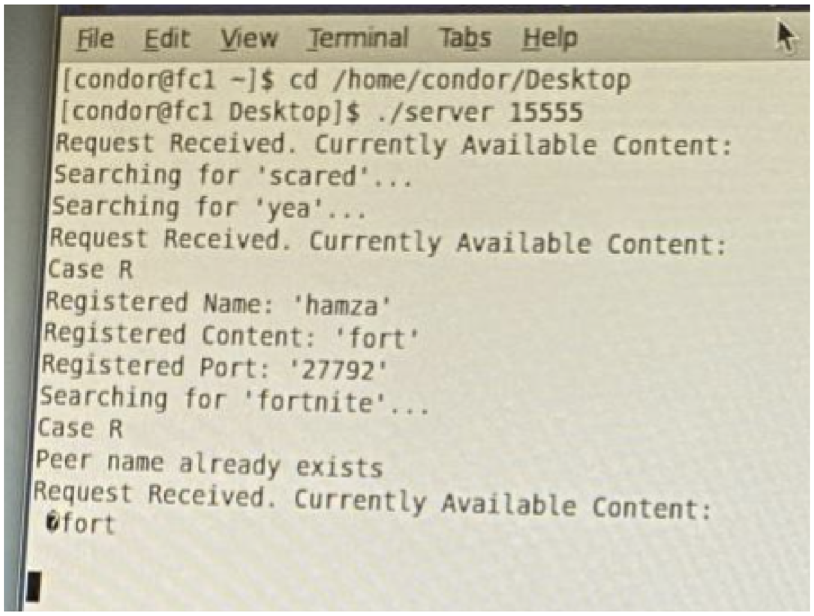
```
Enter Command:
R
Enter content name:
scared
Registration Name: scared
Registration Port: '59562'
```

Moreover, when Peer 1 tries to make a content with the same name again it defaults to an error:



```
Enter Command:
R
Enter content name:
scared
Peer name already exists
```

Next, another peer, (Peer 2) requests a search of a content from the index server and receives the address of a content server. From the client side the Peer enters the command "S" and the server responds with the following output:

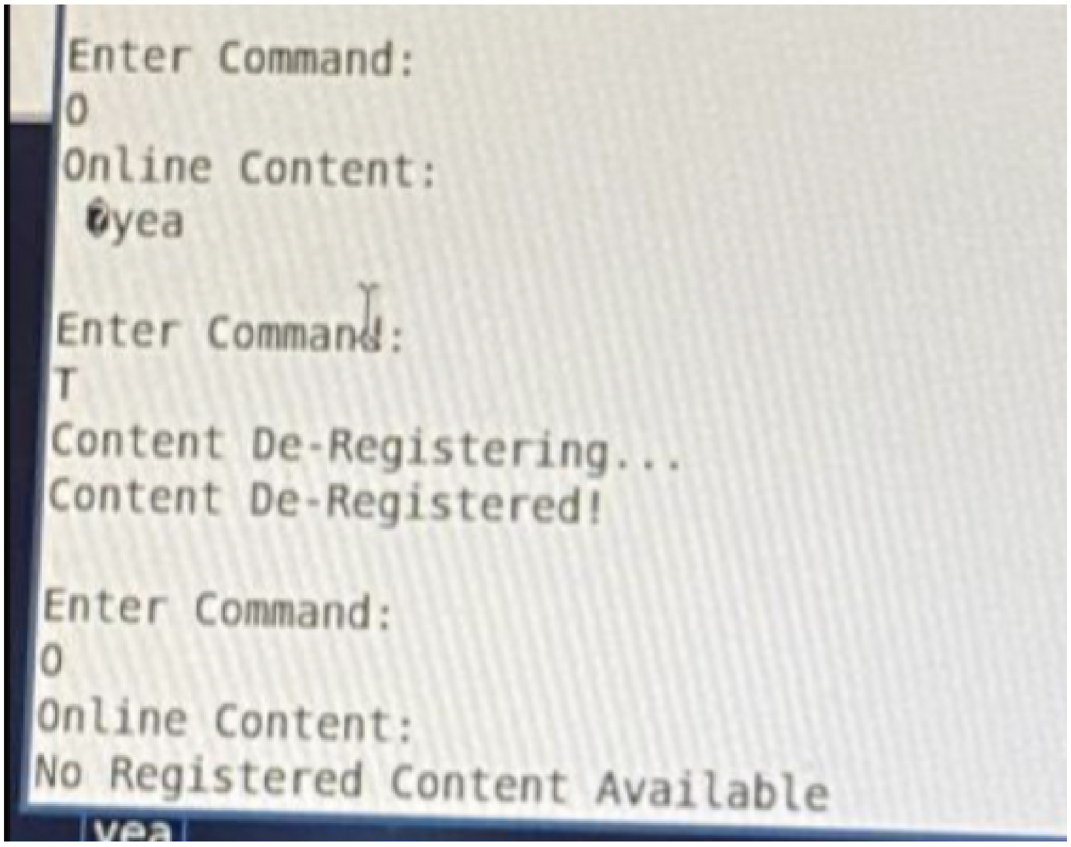


```
[condor@fc1 ~]$ cd /home/condor/Desktop
[condor@fc1 Desktop]$ ./server 15555
Request Received. Currently Available Content:
Searching for 'scared'...
Searching for 'yea'...
Request Received. Currently Available Content:
Case R
Registered Name: 'hamza'
Registered Content: 'fort'
Registered Port: '27792'
Searching for 'fortnite'...
Case R
Peer name already exists
Request Received. Currently Available Content:
 �fort
```

Where, user *ark* registered “yea” and user *hamza* registered “fort”. All the content is registered on a unique port.

Subsequently, Peer 2 uses the address information to download the requested content from the content server. However, this part of the code did not work in practice due to the error “Segmentation Fault”. Thus, Peer 2 was not able to automatically register as a content server of the downloaded content.

Finally, when a given peer wants to quit the program, all the content registered by the peer is deregistered with command “T” as demonstrated in the following code:



```
Enter Command:
O
Online Content:
 yea

Enter Command:
T
Content De-Registering...
Content De-Registered!

Enter Command:
O
Online Content:
No Registered Content Available
```

Conclusion

Through the development of both the client and server programs, this project provided the perfect opportunity to exercise the concepts of TCP, UDP and socket programming. The relationships between TCP and UDP provide a foundation for the success of peer to peer processes. These relationships are commonly used in real-world applications like uTorrent and Bitttorrent. These applications utilize the security that TCP provides and combine it with the UDP logic in communicating from one peer to another. The client and server programs developed in this lab were successfully able to practice all the concepts except for the download component as it ran into a system memory error.

References

1. *Socket programming*. (n.d.). © Copyright IBM Corporation 2001, 2010.

   https://www.ibm.com/docs/en/i/7.1?topic=communications-socket-programming

2. GeeksforGeeks. (2022, September 19). *Socket Programming in C/C++*.

   https://www.geeksforgeeks.org/socket-programming-cc/

3. *TCP vs UDP*. (n.d.). Diffen. https://www.diffen.com/difference/TCP_vs_UDP

4. *P2P_Project2022.* (2022, November 29). Toronto Metropolitan University, COE 768.

   https://courses.ryerson.ca/d2l/le/content/680205/viewContent/4710335/View

Appendix

Server.c:

```
/* time_server.c - main */

#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>


#definePACKETSIZE 101
#defineDATASIZE     100
#defineNAMESIZE     10
#defineADDRSIZE     80
#defineMAXCONTENTPORTS        5
#defineMAXPEERS   5
#defineMAXCONTENTLEN 50
/*------------------------------------------------------------------------
 * main - Index Server
 *------------------------------------------------------------------------
 */
struct content {
	char name[NAMESIZE];
	char user[NAMESIZE];
	int port;
	struct   sockaddr_in addr;
};

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

struct registerPDU {
        char type;
        char peerName[NAMESIZE];
        char contentName[NAMESIZE];
        struct   sockaddr_in addr;
};

int getUserIndex(struct content contents[MAXPEERS], char name[]);
int getContentIndex(struct content contents[MAXPEERS], char name[], int startIndex);
void getContent(struct content contents[MAXPEERS], int length, char *msg);
void removeContent(struct content contents[MAXPEERS], int length, int index);

int main(int argc, char *argv[])
{
        struct  sockaddr_in fsin;       /* the from address of a client*/
        int     sock;                   /* server socket                */
        int     alen;                   /* from-address length                  */
        struct  sockaddr_in sin; /* an Internet endpoint address        */
        int     s, type, n;     /* socket descriptor and socket type    */
        int     port=3000;

        /* PDUs */
        struct  registerPDU     rpdu, dpdu;     // Standard PDU structure
        struct  pdu     spdu;   // Acknowledgement-Errors

        /* Peer-Content Information */
        struct   content *contents = malloc(MAXPEERS * sizeof(struct content));
        int     nextNameIndex = 0, nextContentIndex = 0;
        int     pcIndex = 0;

        /* Messages */
        char    nameExistsMsg[] = "Peer name already exists\n";
        char    noContentMsg[] = "No Registered Content Available\n";
        char    allContentMsg[MAXCONTENTLEN+5];

        int i;

        switch(argc){
                case 1:
                        break;                  // ./server
                case 2:
                        port = atoi(argv[1]);    // ./server 15000
                        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);

        while (1) {
                if ((n = recvfrom(s, &rpdu, sizeof(rpdu), 0, (struct sockaddr *)&fsin, &alen)) < 0)
// Get PDU
                        fprintf(stderr, "recvfrom error\n");

                switch(rpdu.type){
                        case 'R':
                                spdu.type = 'A';

                                /* Check for already existing peer name */
                                if(nextNameIndex    >    0    &&    getUserIndex(contents,
rpdu.peerName) >= 0)
                                {
                                        printf("%s", nameExistsMsg);
                                        spdu.type = 'E';
                                        memcpy(spdu.data,                    nameExistsMsg,
sizeof(nameExistsMsg));
```

```
                        }

                        /* Send acknowledgement or error if peer name already exists */
                        if(sendto(s, &spdu, PACKETSIZE, 0, (struct sockaddr *)&fsin,
sizeof(fsin)) < 0){
                              fprintf(stderr, "Error sending data\n");
                              exit(1);
                        }
                        if (spdu.type == 'E')
                              break;

                        printf("Registered Name: '%s'\n", rpdu.peerName);
                        printf("Registered Content: '%s'\n", rpdu.contentName);
                        printf("Registered Port: '%d'\n", rpdu.addr.sin_port);

                        strcpy(contents[nextContentIndex].name, rpdu.contentName);
// Add new content
                        contents[nextContentIndex].name[NAMESIZE] = '\0';
                        strcpy(contents[nextContentIndex].user, rpdu.peerName);
                        contents[nextContentIndex].user[NAMESIZE] = '\0';
                        contents[nextContentIndex].addr = rpdu.addr;
                        nextContentIndex++;
                        nextNameIndex++;
                        break;
                  case 'O':
                        /* If no content registered yet, send no content available otherwise
send all registered content */
                        printf("Request Received. Currently Available Content: \n");
                        if (nextContentIndex == 0){
                              spdu.type = 'E';
                              memcpy(spdu.data,                    noContentMsg,
sizeof(noContentMsg));
                              if(sendto(s, &spdu, PACKETSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin)) < 0){
                                    fprintf(stderr, "Error sending data\n");
                                    exit(1);
                              }
                        }
                        else {
```

```
                    getContent(contents, nextContentIndex, allContentMsg);
                    spdu.type = 'O';
                    memcpy(spdu.data,                    allContentMsg,
sizeof(allContentMsg));
                    if(sendto(s, &spdu, PACKETSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin)) < 0){
                        fprintf(stderr, "Error sending data\n");
                        exit(1);
                    }
                }
                break;
            case 'T':
                /* If no content registered yet, send no content available otherwise
send all registered content */
                pcIndex = getUserIndex(contents, rpdu.peerName); //    Registered
content is monitored on peer side
                removeContent(contents, nextContentIndex, pcIndex);        //
Remove content from list of registered content
                nextContentIndex--;
                nextNameIndex--;
                spdu.type = 'A';
                if(sendto(s, &spdu, PACKETSIZE, 0, (struct sockaddr *)&fsin,
sizeof(fsin)) < 0){
                    fprintf(stderr, "Error sending data\n");
                    exit(1);
                }
                break;
            case 'S':
                printf("Searching for '%s'...\n", rpdu.contentName);
                pcIndex    =    getContentIndex(contents,    rpdu.contentName,
nextContentIndex); // Check if content registered
                if (pcIndex == -1){
                    dpdu.type = 'E';
                }
                else {
                    printf("'%s' found\n", rpdu.contentName);
                    dpdu.type = 'S';
                    dpdu.addr = contents[pcIndex].addr;
                }
                /* Send address of content server otherwise send error */
```

```
                    if(sendto(s, &dpdu, PACKETSIZE, 0, (struct sockaddr *)&fsin,
sizeof(fsin)) < 0){
                        fprintf(stderr, "Error sending data\n");
                        exit(1);
                    }
                    break;
                default:
                    printf("Case not recognized\n"); // No functionality for received
PDU type
                    break;
            }
            bzero(spdu.data, DATASIZE);
        }
}

/* Get index of existing user */
int getUserIndex(struct content contents[MAXPEERS], char name[])
{
        int i;
        for (i = 0; i < MAXPEERS; i++)
        {
            if (strcmp(contents[i].user, name) == 0)
                    return i;
        }
        return -1;
}

/* Get index of latest content (in case content is not unique) */
int getContentIndex(struct content contents[MAXPEERS], char name[], int startIndex)
{
        int i;
        for (i = startIndex; i >= 0; i--)
        {
            if (strcmp(contents[i].name, name) == 0)
                    return i;
        }
        return -1;
}

/* Get all currently registered content with unique names */
```

```
void getContent(struct content contents[MAXPEERS], int length, char *msg)
{
        char buffer[MAXCONTENTLEN+5];
        int i,j,k;
        char uniqueNames[length][NAMESIZE];
        int nameCount = 1;
        int duplicate = 0;

        strcpy(uniqueNames[0], contents[0].name);
        for(i = 1; i < length; i++) // Start at cell[1] and check for duplicates in the rest of array
        {
                for(j = 0; j < nameCount; j++)
                {
                        if (strcmp(contents[i].name, uniqueNames[j]) == 0)
                                duplicate = 1;
                }
                if (duplicate == 0){    // If unique, copy into array
                        strcpy(uniqueNames[nameCount], contents[i].name);
                        nameCount++;
                }
                duplicate = 0;
        }

        for (k = 0; k < nameCount; k++)
        {
                strcat(buffer, uniqueNames[k]);
                strcat(buffer, "\n");
        }
        printf("%s\n", buffer);
        strcpy(msg, buffer);
}

/* Remove and resize content array */
void removeContent(struct content contents[MAXPEERS], int length, int index)
{
        int i;
        printf("De-Registering Name: '%s'\n", contents[index].user);
        printf("De-Registering Content: '%s'\n", contents[index].name);
        printf("De-Registering Port: '%d'\n", contents[index].addr.sin_port);
        for (i = index; i < length; i++)
        {
                if (i > 0 && i + 1 >= length){
                        strcpy(contents[i].name, "");
                        strcpy(contents[i].user, "");
                        break;
                }
                contents[i] = contents[i+1];
        }
        printf("De-Registered\n");
}
```

Client.c:

```
/* time_client.c - main */

#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/types.h>
#include <sys/stat.h>
#include <errno.h>

#include <netdb.h>
#include <dirent.h>

#defineBUFSIZE 64
#definePACKETSIZE 101
#defineDATASIZE     100

#defineNAMESIZE    10

/*------------------------------------------------------------------------
 * main - PEER
 *------------------------------------------------------------------------
 */

 struct pdu {
        char type;
        char peerName[NAMESIZE];
        char contentName[NAMESIZE];
        struct   sockaddr_in addr;
 };
 struct standardPDU {
        char type;
        char data[DATASIZE];
 };

void handle_user(int s, struct sockaddr_in reg_addr, char *peerName, int *registered);
void handle_client(int sd);
void sendFile(int s, FILE *p, int fileByteSize);
int getContentServer(int s, char peerName[], char downloadName[], struct sockaddr_in addr);
int download_request(int sd, char downloadName[]);
int receiveFile(int sd, char fileName[]);
int getLocalFiles(void);
int setupPDU(struct pdu *spdu, char type, char peerName[], char contentName[]);

int main(int argc, char **argv)
{
        char    *host = "localhost";
        int     port = 3000;
        struct hostent  *phe;   /* pointer to host information entry    */
        struct sockaddr_in indexServer, client, reg_addr;      /* an Internet endpoint address
        */
        int     client_len,     reg_len;
        int     s, type, sd, new_sd;    /* socket descriptor and socket type   */
        int     ret_sel;
        char    peerName[NAMESIZE];
        int     registered = 0;
        int     *regPtr = &registered;
        int     s_counter;      // Socket counter

        switch (argc) {
        case 1:
                break;
        case 2:
                host = argv[1];// ./peer locahost
        case 3:
                host = argv[1];// ./peer locahost 15000
                port = atoi(argv[2]);
                break;
        default:
                fprintf(stderr, "usage: UDPtime [host [port]]\n");
                exit(1);
        }

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

/* Map host name to IP address, allowing for dotted decimal */
    if ( phe = gethostbyname(host) ){
            memcpy(&indexServer.sin_addr, phe->h_addr, phe->h_length);
    }
    else if ( (indexServer.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 *)&indexServer, sizeof(indexServer)) < 0)
                fprintf(stderr, "Can't connect to %s %s \n", host, "Time");

/* Setup TCP socket */
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(EXIT_FAILURE);
        }
        bzero((char*)&reg_addr, sizeof(struct sockaddr_in));
        reg_addr.sin_family = AF_INET;
        reg_addr.sin_port = htons(0);
        reg_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
        if (bind(sd,(struct sockaddr*)&reg_addr, sizeof(reg_addr)) == -1){
                fprintf(stderr, "Can't bind name to socket\n");
                exit(EXIT_FAILURE);
        }

        reg_len = sizeof(struct sockaddr_in);
        getsockname(sd,(struct sockaddr*)&reg_addr,&reg_len);
```

```
/* Queue up to 10 connection requests */
        if(listen(sd,10) < 0){
                fprintf(stderr, "Listening failed\n");
                exit(EXIT_FAILURE);
        }

/* Listen to multiple sockets */
   fd_set rfds, afds;

/* Prompt Peer for Name */
        peerName[0] = '\0';
        printf("Enter username:\n");

        while(1) {

                FD_ZERO(&afds);
        FD_SET(0,&afds);     // Listening on stdin
        FD_SET(sd, &afds);   // Listening on server TCP socket
                memcpy(&rfds, &afds, sizeof(rfds));

                if (ret_sel = select(FD_SETSIZE, &rfds, NULL, NULL, NULL) < 0){
                        printf("Select() Error\n");
                        exit(EXIT_FAILURE);
                }

                if(FD_ISSET(sd, &rfds)) {     // Check server TCP socket
                        client_len = sizeof(client);
                        new_sd = accept(sd,(struct sockaddr*)&client,&client_len);
                        if (new_sd >= 0) {                  // New Accepted TCP socket
                                handle_client(new_sd);          // Handle download request
                                close(new_sd);
                                printf("Enter Command:\n");
                        }
                }
                if(FD_ISSET(fileno(stdin), &rfds)) {
                        handle_user(s, reg_addr, peerName, regPtr); // Handle user interaction
                        printf("Enter Command:\n");
                }
        }
```

```
	close(sd);
	exit(EXIT_SUCCESS);
}

/* Peer to Index Server interactions */
void handle_user(int s, struct sockaddr_in reg_addr, char *peerName, int *registered)
{
	struct pdu spdu;
	struct standardPDU rpdu;
	char	cmd;
	char	contentName[NAMESIZE];
	char	helpMsg[] = "R - Registration, T - De-Registration, D - Download, L - List Local
Content, O - List of On-Line Registered Content, Q - Quit";
	int	cont_server, downloadStatus;
	char	downloadName[NAMESIZE];

	if(peerName[0] == '\0'){
		scanf("%s", peerName);
		peerName[NAMESIZE] = '\0';
		printf("Enter Command:\n");
	}

	scanf(" %c", &cmd);
	switch(cmd){
		case 'D':
			printf("Enter name of content to download:\n");
			scanf("%s", downloadName);
			cont_server = getContentServer(s, peerName, downloadName, reg_addr);
// Ask index server to search for content and receive address of content server
			if (cont_server < 0){
				printf("No such content available\n\n");
				break;
			}
			downloadStatus = download_request(cont_server, downloadName);
// Send download request to content server and receive file
			if (downloadStatus == -1)
				break;
		case 'R':
			if (cmd == 'R'){
				printf("Enter content name:\n");
```

```
                    scanf("%s", contentName);
                    setupPDU(&spdu, 'R', peerName, contentName);
                }
                else {
                    setupPDU(&spdu, 'R', peerName, downloadName);
                }
                spdu.addr = reg_addr;
                write(s, &spdu, sizeof(spdu));// Send registration PDU to index server

                recv(s, &rpdu, PACKETSIZE, 0);      // Receive acknowledgement or error
PDU from index server
                if (rpdu.type == 'E') {
                    printf("%s\n", rpdu.data);
                }
                else if (rpdu.type == 'A') {
                    printf("Registration Name: %s\n", spdu.contentName);
                    printf("Registration Port: '%d'\n\n", spdu.addr.sin_port);
                    ++(*registered);
                }
                break;
            case 'Q':
            case 'T':
                if (*registered == 0 && cmd == 'T') // If user tries to register content
without any registered content
                    printf("You have no registered content\n");
                else if (*registered >= 1){
                    printf("Content De-Registering...\n");
                    setupPDU(&spdu, 'T', peerName, contentName);
                    write(s, &spdu, sizeof(spdu));// Send de-registration PDU to index
server
                    recv(s, &rpdu, PACKETSIZE, 0);      // Receive Acknowledgement
of de-registration
                    if (rpdu.type == 'A') {
                        --(*registered);
                        printf("Content De-Registered!\n\n");
                    }
                }
                if (cmd == 'Q'){         // If quitting, exit
                    printf("Terminating...\n");
                    exit(EXIT_SUCCESS);
```

```
				}
				break;
			case 'O':
				spdu.type = 'O';
				write(s, &spdu, sizeof(spdu));// Request online content from index server
				recv(s, &rpdu, PACKETSIZE, 0);
				printf("Online Content:\n%s\n", rpdu.data);
				break;
			case 'L':
				if(getLocalFiles() == 0)
					printf("No local content available\n");
				break;
			case '?':
				printf("%s\n\n", helpMsg);
				break;
			default:
				break;
		}
}

/* Handle content client download request */
void handle_client(int sd)
{
	struct pdu rpdu;
	struct standardPDU spdu;
	char	fileName[NAMESIZE];
	char	fileNotFound[] = "FILE NOT FOUND\n";
	int	n;
	FILE	*file;

	if ((n = recv(sd, &rpdu, PACKETSIZE, 0)) == -1){
		fprintf(stderr, "Content Server recv: %s (%d)\n", strerror(errno), errno);
		exit(EXIT_FAILURE);
	}
	if (rpdu.type == 'D'){
		memcpy(fileName, rpdu.contentName, NAMESIZE);
		char filePath[NAMESIZE+2];		// Add current directory to file name
		snprintf(filePath, sizeof(filePath), "%s%s", "./", fileName);

		file = fopen(filePath, "r");
```

```
                if (file == NULL) {                     // File does not exist
                        spdu.type = 'E';
                        memcpy(spdu.data, fileNotFound, sizeof(fileNotFound));
                        write(sd, &spdu, sizeof(spdu));
                }
                else {
                        printf("Sending file...\n");
                        struct stat fileInfo;
                        stat(fileName, &fileInfo);
                        sendFile(sd, file, fileInfo.st_size);
                        printf("Successfuly sent file\n\n");
                        fclose(file);
                }
        }
}

/* Send file from content server to content client */
void sendFile(int sd, FILE *p, int fileByteSize)
{
        struct  standardPDU packet;
        char    fileData[DATASIZE] = {0};
        int     n, bytesSent, totalBytesSent = 0;

        while((n = fread(fileData, sizeof(char), DATASIZE, p)) > 0) {       //    NULL    if    EOF
reached or error occurs
                if (totalBytesSent + DATASIZE >= fileByteSize)
                        packet.type = 'F';
                else
                        packet.type = 'C';

                memcpy(packet.data, fileData, DATASIZE);

                if((bytesSent = send(sd, &packet, sizeof(packet), 0)) == -1){
                  fprintf(stderr, "Error sending data\n");
                  exit(1);
                }
                totalBytesSent += n;
                bzero(fileData, DATASIZE);  //Erase data
        }
}
```

```
/* Get content server info from index server and connect to socket */
int getContentServer(int s, char peerName[], char downloadName[], struct sockaddr_in addr)
{
        struct pdu spdu, rpdu;
        struct standardPDU dpdu;
        int tcp_sock;

        setupPDU(&spdu, 'S', peerName, downloadName);
        spdu.addr = addr;
        write(s, &spdu, sizeof(spdu));
        recv(s, &rpdu, PACKETSIZE, 0);     // Receive content server address/port from index
server

        if(rpdu.type == 'E'){   // If error, content doesn't exist
                return -1;
        }
        else if (rpdu.type == 'S')
                setupPDU(&spdu, 'D', peerName, downloadName);

        printf("Address Received. Establishing Connection With Content Server...\n");
        printf("Content Server Port: '%d'\n", rpdu.addr.sin_port);

        /* Create a socket */
    if ((tcp_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1){
        fprintf(stderr, "Can't create a socket\n");
        exit(EXIT_FAILURE);
    }

        /* Connect to the content server */
    if (connect(tcp_sock, (struct sockaddr *)&rpdu.addr, sizeof(rpdu.addr)) == -1){
        fprintf(stderr, "Can't Connect: %s (%d)\n", strerror(errno), errno);
        exit(EXIT_FAILURE);
    }

    printf("Content Server Connection Established!\n");
    return tcp_sock;
}

/* Send download request to content server and wait to receive file */
```

```
int download_request(int sd, char downloadName[]){
        struct pdu spdu;
        int bytes, ret = -1;

        spdu.type = 'D';
        strcpy(spdu.contentName, downloadName);
        spdu.contentName[NAMESIZE] = '\0';
        bytes = write(sd, &spdu, sizeof(spdu));
        ret = receiveFile(sd, downloadName);

        return ret;
}

/* Receive file data from content server */
int receiveFile(int sd, char fileName[])
{
        int     n;
        FILE    *file;
        struct standardPDU packet;

        file = fopen(fileName, "w");  // Create file
        if (file == NULL) {
                fprintf(stderr, "Can't create file \n");
                return -1;
        }

        while (1) {
                n = recv(sd, &packet, PACKETSIZE, 0);
                packet.data[DATASIZE] = '\0';
                if (packet.type == 'E'){
                        printf("ERROR: %s\n", packet.data);
                        remove(fileName);
                        return -1;
                }
                fprintf(file, "%s", packet.data);         // Write to file
                if (packet.type  == 'F'){                 // When final pdu received, break loop
                        break;
                }

        }
```

```
	fclose(file);
	return 0;
}

int getLocalFiles(void)
{
	DIR *currDir;
	struct dirent *entry;
	currDir = opendir("."); // Open current directory
	int numberOfContent = 0;

	if(currDir){
		while ((entry = readdir(currDir)) != NULL)
		{
			if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
				continue; // Disclude these names
			if (entry->d_type == DT_REG){
				const char *extension = strrchr(entry->d_name,'.');
				if (extension != NULL && strcmp(extension, ".c") != 0){
					numberOfContent++;
					printf("%s\n", entry->d_name);
				}
			}
		}
		closedir(currDir);
	}
	return numberOfContent;
}

/* Generic PDU structure mapping */
int setupPDU(struct pdu *spdu, char type, char peerName[], char contentName[])
{
	spdu->type = type;
	strcpy(spdu->peerName, peerName);
	spdu->peerName[NAMESIZE] = '\0';
	strcpy(spdu->contentName, contentName);
	spdu->contentName[NAMESIZE] = '\0';
	return 0;
}
```