Ryerson University

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

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

Instructor	Dr. Cungang (Truman) Yang

Assignment/Lab Number:	n/a
Assignment/Lab Title:	P2P Project

Submission Date:	November 30th, 2023
Due Date:	December 1st, 2023

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Ranjith	Ashanth	500960721	10	Ashanth
Soubra	Karim	500966625	10	KS
Morra	Fabiano	500950505	10	

*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. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Introduction

A P2P program is developed in the final project of the course. The P2P program has two components, a peer and an index server. The peer can be both a client or server depending on the current functionality, it can also communicate with other peers or the index server. Among the peer's functions are to register content , download content , and host content. The index server's main function is to keep track of which peer contains which content. The two main types of communication between the peers and index server are UDP and TCP. UDP is used to communicate between the peer and index server ,and TCP is used to communicate between peers. The UDP connection can either be connected or connectionless, UDP prioritizes speed over reliability. In the P2P program , the peer has a connected UDP connection while the index server has a connectionless UDP connection, this is due to the index server needing to communicate with multiple peers while there is only one index server. TCP requires a handshake between the client and server in order to establish a connection.The communication between peers is TCP since TCP is more reliable than UDP , it can be used for file downloads. Since a peer can be both a server and client , the peer that is hosting the file that is to be downloaded is the server , meanwhile the peer that is requesting the file is would be set up as a client. Since this is a P2P program , the peer that requested the file then becomes a host once the requested file is downloaded.

Description of Client and Server Programs

In this project, there will be 2 different programs, the peer which can act as the content client (downloads content from another peer) or the content server (stores the content available for download) along with the server which helps to facilitate communication between the peers as a central point. The peer leverages the select() method to listen to multiple sockets and standard input by the user so that it can respond to download requests by other peers and also handle the user's input. The peer will utilize UDP to connect with the server, and will take inputs from the user to register content, view registered content, download content, and deregister content while the server will handle storing all the registered content and their respective location information such as the peer, IP address, and port number. These functionalities will be described in further detail below.

Peer (Client)

Content Registration

In content registration, we make a request to the index server to register content and make the content available for download by other peers. We do this by making a request with the type 'R' along with the peer name, content name, IP address, and port number, so that another peer will know how to connect to our server and download the content. Before creating the request however, we must create a TCP socket so that other peers can connect, and retrieve the IP address and port number from this TCP socket. We can now make the request to the index server and then wait for the acknowledgement that the content has been successfully registered. Once successfully registered, we can add the socket descriptor to the FD_SET, so that we are listening for any download requests from other peers and also add the content name and socket descriptor to our local list for keeping track of listings specific to the peer that we are on. Figures 1 and 2 show the implementation of the registration

function in the peer. In figure 2, the peer sends a type R PDU and receives a type A PDU to confirm registration.

```c
void registration(int s_sock, char *name, struct sockaddr_in server)
{
    /* Register a content to the index server
        - Create a TCP socket for other peers to download
        - Add TCP socket descriptor and content name to local list
        - Register the content to the index server via UDP
        - Update nfds (number of file descriptors)
    */
    int s;
    // Create a stream socket
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    { // Create TCP socket for content
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    // Bind an address to the socket
    struct sockaddr_in reg_addr; // TCP Socket Address
    bzero((char *)&reg_addr, sizeof(struct sockaddr_in));
    reg_addr.sin_family = AF_INET;
    reg_addr.sin_port = htons(0); // Is 0 correct?
    reg_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) == -1)
    {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }

    // Queue upto max 5 connect requests
    listen(s, 5);

    char port[10];
    int alen = sizeof(struct sockaddr_in);
    getsockname(s, (struct sockaddr *)&reg_addr, &alen); // Get socket address information
    sprintf(port, "%d", ntohs(reg_addr.sin_port));       // Get port of socket and put into string
```

Figure 1. Registration code implementation 1

```c
    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'R';
    strcat(rpdu->data, name);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, usr);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, inet_ntoa(reg_addr.sin_addr));
    strcat(rpdu->data, "|");
    strcat(rpdu->data, port);
    // printf("Registering peer %s \n", usr);
    printf("Registration data sent to index server: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server, sizeof(server));

    // Recieve acknowledgement of content registration
    PDU recv_pdu;
    int i, n, len;
    recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server, &len);

    if (recv_pdu.type == 'A')
    {
        strcpy(table[max_index].name, name);
        int i;
        for (i = 0; i < max_index; i++)
        {
            if (strcmp(table[i].name, "") && s >= nfds)
            {
                nfds = s + 1;
            }
        }
        table[max_index].sd = s;
        FD_SET(s, &afds);
        max_index++;
    }
    else
    {
        printf("Error: %s\n", recv_pdu.data);
    }
}
```

Figure 2.Registration code implementation 2.

Content Listing

In content listing, we make a request to the index server to see all currently registered content. This request will use the type O, and will receive a string containing the content names separated by a pipe. The function will break the string into a series of tokens using the pipe delimiter and print out all the registered content to the console for the user. Figure 3 shows the implementation of the content listing feature. The Peer sends a type O PDU and receives a list of content registered with the peer.

```
void online_list(int s_sock, struct sockaddr_in server)
{
    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'O';
    strcat(rpdu->data, "send list please");

    // printf("Registering peer %s \n", usr);
    // printf("Data sent for online list: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server, sizeof(server));

    // Recieve acknowledgement of content registration
    PDU recv_pdu;
    int i, n, len;
    char listed_files[100];
    char *token;
    int index=0;
    n=recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server, &len);
        if (recv_pdu.type == 'O'){
            printf("Received registered files from index server: \n");
            token = strtok(recv_pdu.data, "|");
            while (token != 0)
            {
                printf("[%d] %s \n", index, token);


                token = strtok(0, "|");
                index++;
            }
       }else{
            printf("error listing \n");
        }

}
```

Figure 3. Content listing implementation.

Content Download

In content download the peer hosting the content is the server and the peer requesting the content is the client. The peer server has multiple TCP sockets open , the number of sockets is dependent on the number of content registered. This enables the peer server to host multiple files at the same time and respond to multiple requests. The select function is used in order to determine which file is requested for download on the peer server side, and it enables the peer server to listen to multiple sockets at the same time . The select function takes in a list of socket/file descriptors and keeps monitoring the socket/file descriptors given to it. The select function is blocking, it keeps blocking until an event occurs in any of the sockets it is listening to , and FD_ISSET is used to determine which socket did the event occur in. Since each socket is tied to a unique piece of content in the peer server, the peer server can determine the file requested depending on the socket the event occurred in. To initiate the download , the peer client sends a S-type PDU to the index server . The S-type PDU contains the content name that the peer client wishes to download, and the index server replies with the address the content is located in . Once the peer client receives the address of the peer server from the index server , the client sends a D-type PDU that contains the peer client's address and port ,as well as the content name. The peer server receives the request from the peer client , and it uses

FD_ISSET and SELECT to determine the content it must send to the peer client. The peer server sends the data in 100 byte increments along with the C-type PDU. Finally, the peer client registers itself , along with the new content it has downloaded, as a new peer server once it has completed downloading the content from the original peer server. Figure 4 shows the server download implementation, if an Error occurred , the peer server will send a type E PDU. If the server download successfully sends the content, a type C PDU will be sent.

```
    n = read(new_sd, recv, sizeof(PDU));
    printf("bytes received %d\n",n);
    printf("CONNECTION TO SERVER DOWNLOAD SUCCESS! \n");
    printf("type received is %c \n",recv->type);
    printf("data is %s\n",recv->data);
    if (recv->type == 'D')
    {
        token = strtok(recv->data, "|");
        while (token != 0)
        {
            if (index == 1)
            {
                strcpy(fname, token);
            }

            token = strtok(0, "|");
            index++;
        }
    }

printf("Name of file to open is: %s \n",fname);
file_pointer_server = fopen(fname, "r");
PDU resp;
if (file_pointer_server == NULL)
{
    printf("file failed to open");
    resp.type = 'E';
    // resp.data = "FILE NOT FOUND";
    strcpy(resp.data, "FILE NOT FOUND");
    write(new_sd, &resp, sizeof(resp));
}
else
{
    while (fgets(resp.data, 100, file_pointer_server) != NULL)
    {
        resp.type = 'C';
        write(new_sd, &resp, sizeof(resp));
    }
    fclose(file_pointer_server);
}
(void)close(new_sd);
```

Figure 4. Peer server content download.

Figure 5 shows the peer client content download, The peer client content download receives data from the peer server and checks the PDU type. If the PDU is type C, the data is written to the newly downloaded file or else if the PDU type received is E this indicates an error occurred and the peer client download will stop immediately.

```
printf("Sending download request to content server with data: %s\n", spdu.data);
printf("with type %c \n",spdu.type);
write(sd, &spdu, sizeof(PDU));
// content server replies with C type
PDU rpdu;
FILE *file_pointer;
file_pointer = fopen(name, "w");
while (n = read(sd, &rpdu, sizeof(struct pdu)) > 0)
{
    if (rpdu.type == 'C')
    {
        fputs(rpdu.data, file_pointer);
    }
    else if (rpdu.type == 'E')
    {
        printf("ERROR FILE DOWNLOAD FAILED \n");

        fclose(file_pointer);
        remove(name);
        return 1;
    }
}

fclose(file_pointer);
```

Figure 5. Peer client content download.

Figure 6 shows the implementation of the peer content search. The content search function sends the S type PDU in order to obtain the location of the content from the index server. Upon receiving the location of the content, the peer client download will commence.

```
PDU *spdu = malloc(sizeof(PDU));
bzero(spdu->data, 100);

spdu->type = 'S';
// strcat(spdu->type, 'S'); // type
strcat(spdu->data, usr); // peer name
strcat(spdu->data, "|");
strcat(spdu->data, name); // content name
printf("Search data sent to index server: %s\n", spdu->data);
printf("type is: %c \n", spdu->type);
sendto(s_sock, spdu, sizeof(*spdu), 0, (const struct sockaddr *)&server, sizeof(server));
int len;
PDU *recv_pdu;
int alen = sizeof(server);
recvfrom(s_sock, rpdu, sizeof(PDU), 0, (struct sockaddr *)&server, &alen);
printf("Search data received from index server: %s\n", rpdu->data);

if (rpdu->type == 'E')
{
    printf("Error from search request: %s\n", rpdu->data);
    return 1;
}
else
{
    // type S
    printf("File found! \n");
    printf("Data Received: %s \n", rpdu->data);
    return 0;
}
```

Figure 6. Peer content search.

Content Deregistration

In content deregistration, a peer makes a request to the index server to deregister content. The peer sends the peer name, filename and data type ‘T’ to the index server. Once acknowledgement has been received from the index server, the socket which was created with that content is closed and the filename is removed from the local list of content. If the inputted filename by the peer is not a filename

belonging to registered content, then the application will ask the user to enter another filename. If the server sends back an error with data type ‘E’, it will output an error message to the terminal. Peers also have the option of using “quit”. The quit function performs the deregister action for each piece of content it has registered. Once all content has been deregistered, the connection between the peer and the index server is closed, and the application closes. Figure 7 shows the implementation of the content deregistration function , this function is used to deregister content and implement the quit function that is found in figure 8.

```c
void deregistration(int s_sock, char *name, struct sockaddr_in server)
{

    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'T'; //De register a file name
    strcat(rpdu->data, name);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, usr);
    strcat(rpdu->data, "|");
   // strcat(rpdu->data, inet_ntoa(reg_addr.sin_addr));
    //strcat(rpdu->data, "|");
    //strcat(rpdu->data, port);
    printf("Deregistering peer %s \n",usr);
    printf("Data Sent: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server, sizeof(server));


 // Recieve acknowledgement of content deregistration
    PDU recv_pdu;
    int i, n, len, s;
    recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server, &len);

    if (recv_pdu.type == 'A')
    {
    printf("File has been deregistered %s \n",name);
        strcpy(table[max_index].name, name);
    int i;
        for (i = 0; i < max_index; i++)
        {
            if (strcmp(table[i].name, name) && s >= nfds) //This finds the filename from the struct a
            {
        strcpy(table[i].name, ""); //This removes file from struct
                nfds = s + 1;
        table[i].sd = s;
        FD_CLR(s, &afds); //This clears socket
        table[i].sd = -1;           //Remove socket from struct
        close(s);               //Close TCP connection
            }
        }

    }
    else
    {
        printf("Error : %s\n", recv_pdu.data);
    }
}
```

Figure 7. Deregistration.

```c
void quit(int s_sock, struct sockaddr_in server)
{
char dereg[10];
    printf("Quitting server, deregistering all content...\n");
    int i;
    for (i = 0; i < max_index; i++){
        strcpy(dereg, table[i].name);
        printf("Degresitering: %s\n", dereg);
        deregistration(s_sock, dereg, server);
    }
    printf("Done\n");
    max_index = 0;
    exit(0);
}
```

Figure 8. Quit peer function.

Index Server (Server)

Content Registration

In content registration, we receive a request to register content from the peer. In this request, we will receive the type ‘R’, along with the peer name, content name, IP address, and port number. We utilize strtok to parse the data using the delimiter ‘|’ sent from the peer. This will be stored in a list so that when another peer wants to download a piece of content, we can send this information so that the content can be downloaded. We will make sure that there is not an existing peer and content already registered before adding it to our list, and if it is successful, we will send back an acknowledgement with type ‘A’ to tell the peer that the content has been registered successfully. Figures 9 and 10 show the implementation of the registration function in the index server side.

```c
void registration(int s, char *data, struct sockaddr_in addr) {
    printf("Data Recieved from Registration: %s\n", data);

    char *sep = strtok(data, "|"); // Content Name
    char contentName[20];
    strcpy(contentName, sep);
    strcpy(list[max_index].name, contentName);

    char peer[20]; // Peer Name
    sep = strtok(NULL, "|");
    strcpy(peer, sep);

    int i = 0;
    int found = 0;
    // Find out if there is existing content and peer already registered
    while (!found && i < max_index) {
        if (!isEntryNULL(i) && strcmp(contentName, list[i].name) == 0 && strcmp(peer, list[i].head->usr) == 0) {
            printf("Found existing content: %s", contentName);
            // Need to add new content server to content
            found = 1;
        } else {
            // No match found yet
            // printf("No match between %s and %s, %s, and %s\n", contentName, list[i].name, peer, list[i].head->usr);
        }
        i++;
    }
```

Figure 9. Registration index server implementation 1.

```
    if (found == 1) {
        // Found an existing content and peer already registered, so sending an error
        PDU *epdu = malloc(sizeof(PDU));
        bzero(epdu->data, 100);
        epdu->type = 'E';
        strcpy(epdu->data, "Peer and file already registered");
        sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr, sizeof(addr));
    } else {
        ENTRY *entry = malloc(sizeof(ENTRY));
        list[max_index].head = entry;
        list[max_index].head->downloads = 0;
        strcpy(entry->usr, peer);

        sep = strtok(NULL, "|"); // IP Address
        inet_aton(sep, (struct in_addr *)&entry->addr.sin_addr);

        int portNum;
        sep = strtok(NULL, "|");  // Port Number
        sscanf(sep, "%d", &portNum);
        entry->addr.sin_port = htons(portNum);

        printf("Port Number: %d\n", portNum);
        printf("Content Name: %s\n", list[max_index].name);
        printf("Peer Name: %s\n", entry->usr);
        printf("Port %d\n", ntohs(entry->addr.sin_port));

        max_index++;

        PDU *apdu = malloc(sizeof(PDU));
        bzero(apdu->data, 100);
        apdu->type = 'A';
        strcpy(apdu->data, "File has been registered!");
        sendto(s, apdu, sizeof(*apdu), 0, (const struct sockaddr *)&addr, sizeof(addr));
    }
}
```

Figure 10. Registration index server implementation 2.

Content Search

In content search, we receive a request by the peer to find the address of a specific content. In this request, we will receive the type ‘S’ along with the content name. We will find a content server which houses the requested content, and if there are multiple content servers for a specific content, the content server with the least downloads will be returned back to the peer. If there is registered content as requested by the peer, then an error will be returned. This feature is usually not used alone, and rather used with respect to the content download feature, so the peer knows the location of the content.Figure 11 shows the implementation of the content search in the index server.

```
if (found == 1) {
    printf("Found registered content");
    PDU *spdu = malloc(sizeof(PDU));
    bzero(spdu->data, 100);
    spdu->type = 'S';
    strcat(spdu->data, list[found_index].name);
    strcat(spdu->data, "|");
    strcat(spdu->data, list[found_index].head->usr);
    strcat(spdu->data, "|");
    strcat(spdu->data, inet_ntoa(list[found_index].head->addr.sin_addr));
    strcat(spdu->data, "|");
    char port[10];
    sprintf(port, "%d", ntohs(list[found_index].head->addr.sin_port));
    strcat(spdu->data, port);
    printf("Search data to be returned to peer: %s \n", spdu->data);
    sendto(s, spdu, sizeof(*spdu), 0, (const struct sockaddr *)&addr, sizeof(addr));
    list[found_index].head->downloads += 1;
} else {
    PDU *epdu = malloc(sizeof(PDU));
    bzero(epdu->data, 100);
    epdu->type = 'E';
    strcpy(epdu->data, "Content not found!");
    sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr, sizeof(addr));
}
// Print contents of list
```

Figure 11. Content Search index server implementation.

Content Deregistration

In content deregistration, the index server receives the peer name and the filename for the specific content to deregister. The server indexes through each registered content until it finds the one with the matching filename and peername. That content is removed from the list and the index server sends back an acknowledgement with type ‘A’ to inform the peer that the registration was successful. If the server is unable to find the content to deregister, the server sends back an error with type ‘E’ which informs the peer that the content could not be deregistered. Figure 12 shows the implementation of the deregistration function in the index server.

```
void deregistration(int s, char *data, struct sockaddr_in addr) {
printf("Data Recieved from deregistration: %s\n", data);
    char *sep = strtok(data, "|"); // Content Name
    char contentName[20];
    strcpy(contentName, sep);
    strcpy(list[max_index].name, contentName); //Might not need this

    char peer[20]; // Peer Name
    sep = strtok(NULL, "|");
    strcpy(peer, sep); //Variable for peer name, important

    int i = 0;
    int found = 0;


    while (!found && i < max_index) {
        if (!isEntryNULL(i) && strcmp(contentName, list[i].name) == 0 && strcmp(peer, list[i].head->usr) == 0) {
            printf("Found content to deregister: %s\n", contentName);
            // Found the content which is registered, need to remove it
        strcpy(list[i].name, ""); //This should replace that content with NULL, therefore removing it
            found = 1;
        } else {
            printf("Content not registered %s and %s\n", contentName, list[i].name);
        }
        i++;
    }
    if (found == 1) {
    //This is where we remove that filename from the list change from adding to removing omehow?

        //max_index++; dont change the max index when removing, just remove form the list just ignore for now

        PDU *apdu = malloc(sizeof(PDU));
        bzero(apdu->data, 100);
        apdu->type = 'A';
        strcpy(apdu->data, "File has been deregistered!\n");
        sendto(s, apdu, sizeof(*apdu), 0, (const struct sockaddr *)&addr, sizeof(addr));

    } else {

    //Final code if file is not registered
    PDU *epdu = malloc(sizeof(PDU));
        bzero(epdu->data, 100);
        epdu->type = 'E';
        strcpy(epdu->data, "File is not registered, therefore cannot be removed");
        sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr, sizeof(addr));
    }
}
```

Figure 12. Index server deregistration.

Observations and Analysis

Figure 13 shows the peer registering its content in the index server. When first executing ,the index server is empty and does not contain information about other peers and its contents. Figure 13 shows peer1 , register rand1.txt with the index server. Peer 1 sends a R type PDU with the following content peer name, content name , and address. The example in figure 13 shows that the peer name is ash, the content name is rand1.txt ,and the address is made up of the ip address and port. The index in figure 13 also shows it receiving the registration request from p1 since it outputs the p1’s information on the index server side , along with the data type R. P1 can now serve as a peer server for rand1.txt. Therefore all subsequent downloads of rand1.txt will be downloaded from p1 until p1 deregisters itself from the peer server.

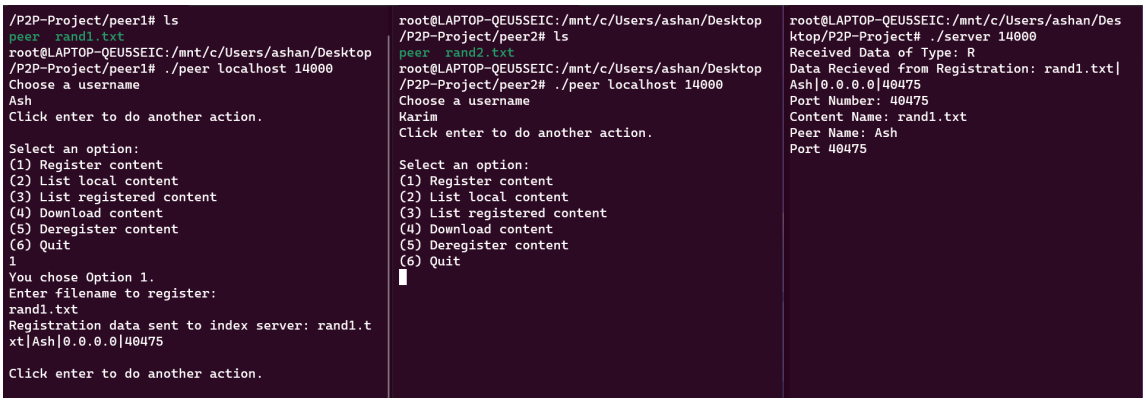

Figure 13. Content registration.

Figure 14 shows the online listing feature's output. P2 checks the content listing before requesting download. P2 sends O-type PDU , which can be seen in the index server output in figure 14. When the index server receives the O-type PDU , the index server replies with all the content names that are registered with the index server. If a content name is deregistered then it will not appear in the content listings if there is no peer associated with the following content name. The output of the content listing changes depending on the content names registered with the index server.

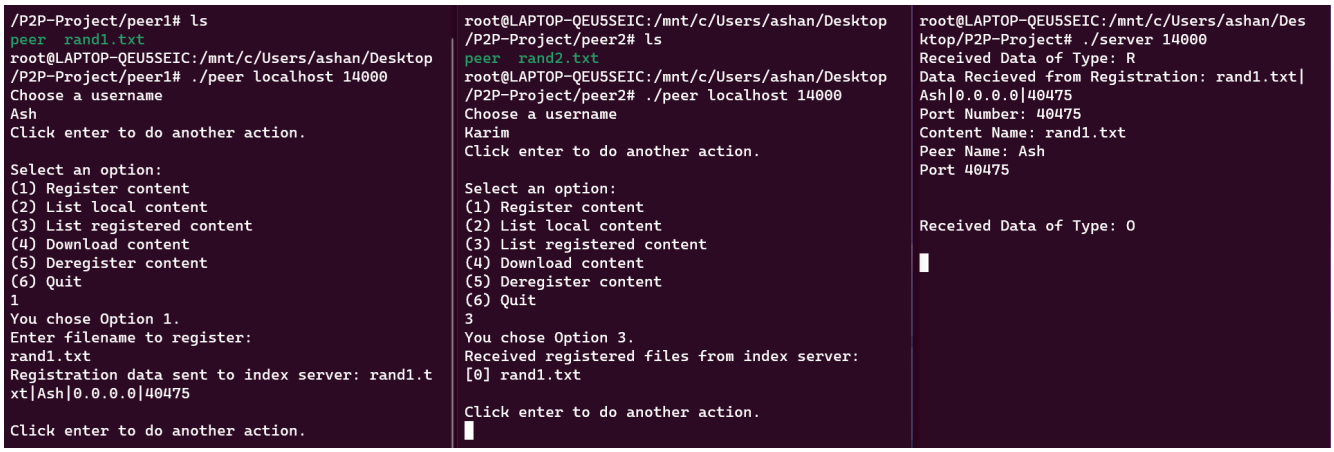

Figure 14. Content listing.

Figure 15 shows the online listing output when more than one file is registered. P1 requests an online content listing from the index server and the registered content in the index server is returned. P1 outputs two files that are registered.

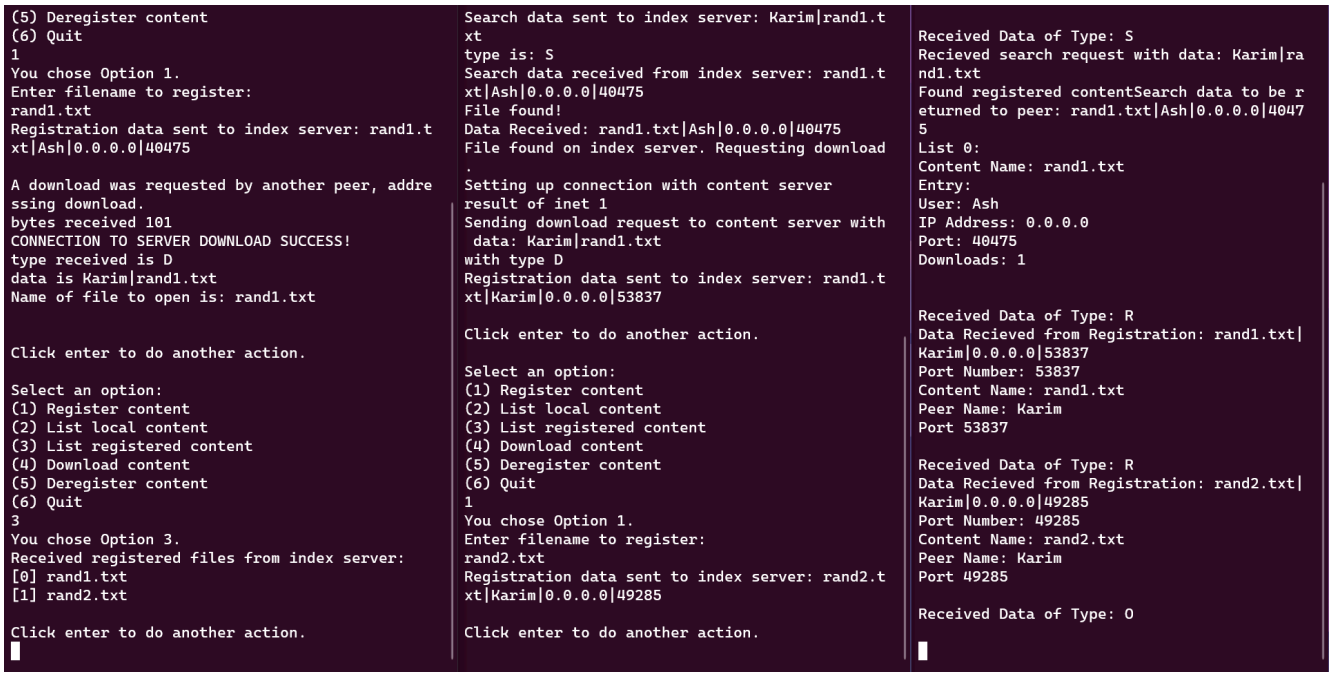

Figure 15. Online listing multiple files.

Figure 16 shows the content download functionality. P2 requests to download rand1.txt . The request first goes to the index server , where the index server replies with the corresponding ip address and port of the peer server that hosts the specific file. After P2 receives the ip address and port number of the location of the file it wants to download , it will connect to the associated peer server to commence the file download. In figure 16, P1 receives the request from P2 , and P1 starts sending the file over the TCP connection to P2. Once the file download is complete , P2 registers itself as a new host for the newly downloaded file and both P1 and P2 are now the host associated with the rand1.txt file . New peers wanting to download rand1.txt can now download the file from P1 or P2.

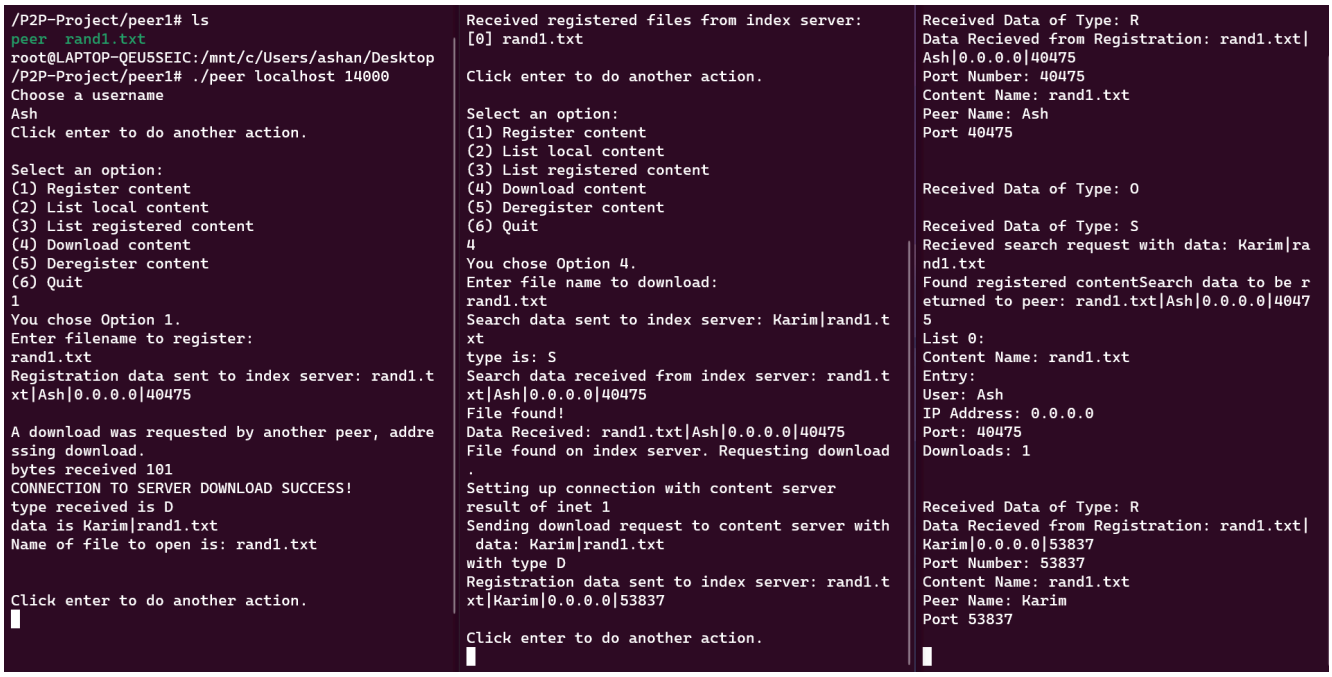

Figure 16. Client - server download.

As seen in Figure 17, the peer makes a request to deregister a piece of content. The client inputs a filename and the data is sent to the server. The server receives the data with type ‘T’ and begins to deregister. Once the content is deregistered, the peer receives confirmation that the content has been deregistered. This is proven in Figure 17 when listing content afterwards, the hello.txt file is no longer on the list of registered content. Figure 18 shows the quit function, which deregistered all content associated with the peer. In this case, 2 items are deregistered which can be proven in Figure 17 due to 2 items being sent to the server with data type ‘T’. Once both items have been deregistered, the peer disconnects from the server.

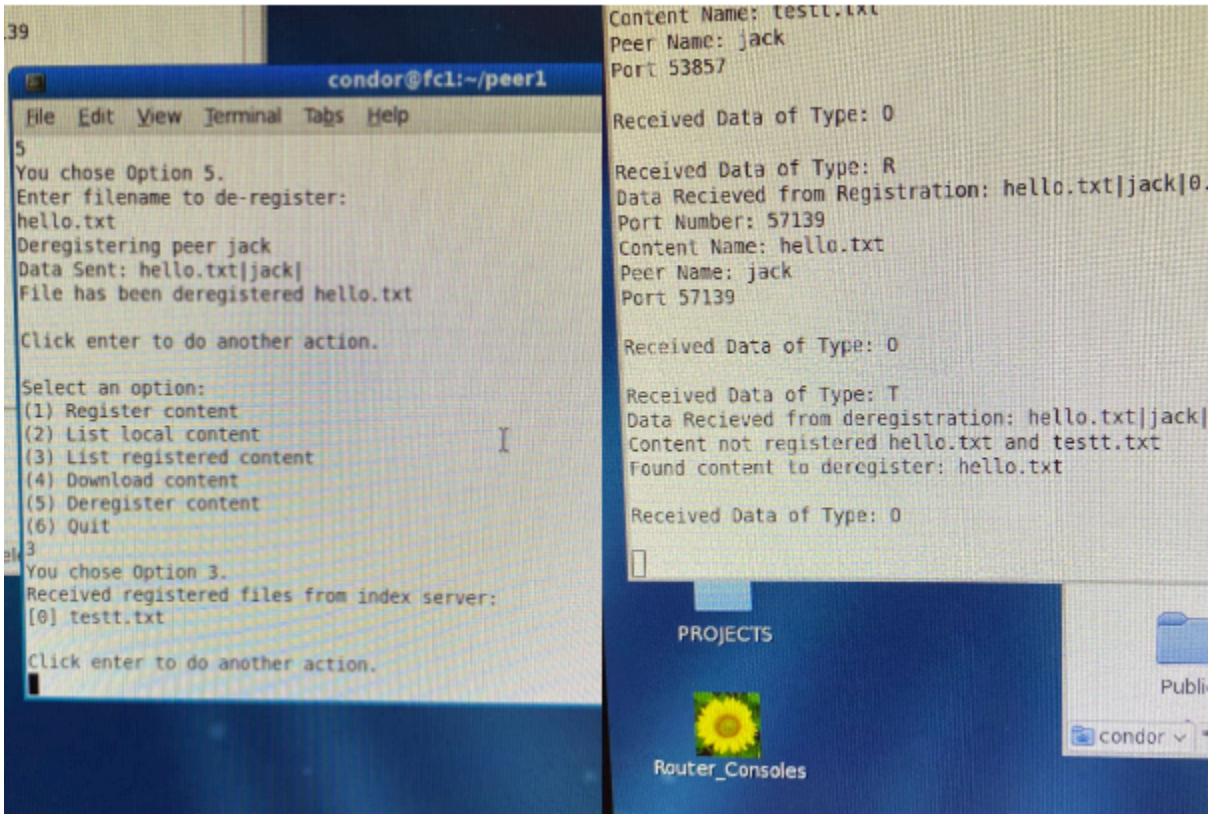

Figure 17. Deregistration function.

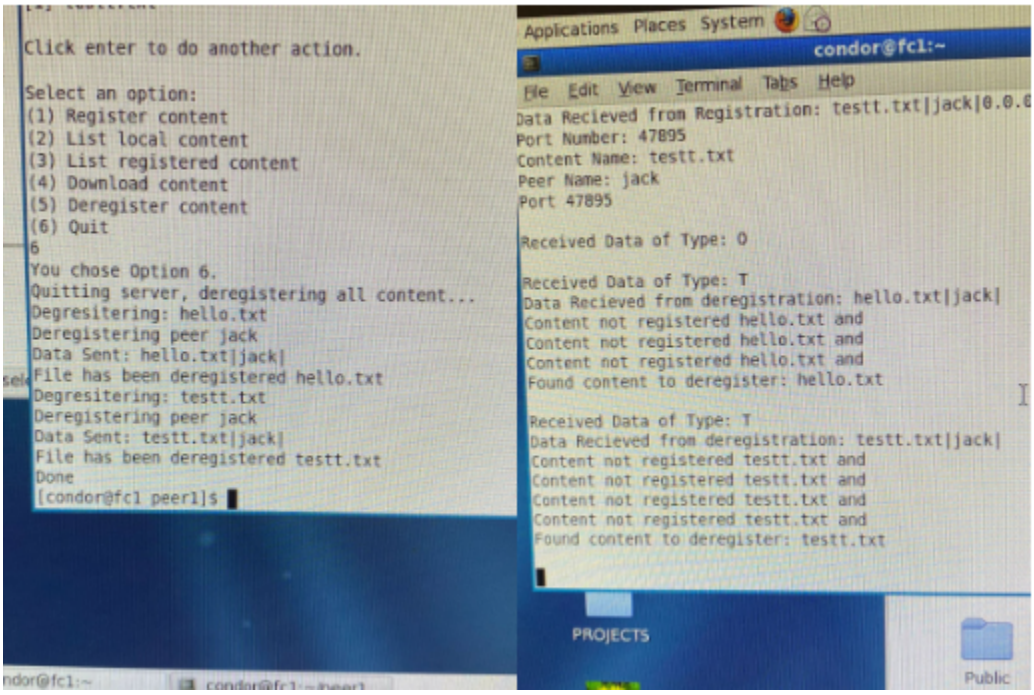

Figure 18. Quit function.

Conclusion

In conclusion, this project was very successful, as we were able to fully build a P2P application using C which was able to register, download, and deregister content. We were able to combine our prior knowledge of UDP and TCP from previous labs and also learn about the select() method for listening to multiple sockets which was needed so that the peer can handle the user's input along with the multiple TCP sockets that was created for each content. This project was a great learning experience, and enhanced our knowledge of computer networking, and also gave us the opportunity to implement that knowledge into a fully working P2P application.

Appendix (Code):

Index server:

```
/* Index Server
Message types:
R - used for registration
A - used by the server to acknowledge the success of registration
Q - used by chat users for de-registration
D - download content between peers (not used here)
C - Content (not used here)
```

```
S - Search content
L - Location of the content server peer
E - Error messages from the Server
*/

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

#define MSG1 "Cannot find content"
#define BUFLEN 100
#define NAMESIZ 20
#define MAX_NUM_CON 200

typedef struct entry {
    char usr[NAMESIZ]; // Peer Name
    struct sockaddr_in addr; // Address of content
    int downloads; // Number of downloads
    struct entry *next; // Other Content Servers
} ENTRY;

typedef struct {
    char name[NAMESIZ]; // Content Name
    ENTRY *head;
} LIST;
LIST list[MAX_NUM_CON];
int max_index = 0;

typedef struct {
    char type;
    char data[BUFLEN];
} PDU;
PDU tpdu;

void search(int, char *, struct sockaddr_in);
void registration(int, char *, struct sockaddr_in);
```

```c
void deregistration(int, char *, struct sockaddr_in);
int isEntryNULL(int);

int main(int argc, char *argv[]) {
    int port = 3000;
    struct sockaddr_in sin; // Server socket
    struct sockaddr_in fsin; // Peer socket
    int alen = sizeof(struct sockaddr_in); // Address length
    char name[NAMESIZ], usr[NAMESIZ];
    PDU rpdu;

    int n;
    for (n = 0; n < MAX_NUM_CON; n++)
        list[n].head = NULL;

    switch (argc) {
        case 1:
            break;
        case 2:
            port = atoi(argv[1]);
            break;
        default:
            fprintf(stderr, "usage: index_server [port]\n");
    }

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

    // Setup UDP socket
    int s;
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0) {
        printf("Can't create socket");
        exit(1);
    }

    // Bind the socket
    if (bind(s, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        printf("Failed to bind socket to port");
```

```
    }

    while (1) {
        if ((n = recvfrom(s, &rpdu, sizeof(PDU), 0, (struct sockaddr *)&fsin,
&alen)) < 0) {
            printf("recvfrom error: n=%d\n", n);
        }

        printf("Received Data of Type: %c\n", rpdu.type);

        if (rpdu.type == 'R') {
            registration(s, rpdu.data, fsin);
        } else if (rpdu.type == 'S') {
            search(s, rpdu.data, fsin);
        } else if (rpdu.type == 'O') {
            // Read from content list and send to client
            PDU *opdu = malloc(sizeof(PDU));
            bzero(opdu->data, 100);
            opdu->type = 'O';
            char content[100];
            bzero(content, 100);
        int i;
            for (i = 0; i < max_index; i++) {
                if (!isEntryNULL(i) && strstr(content, list[i].name) == NULL)
{
                    strcat(opdu->data, list[i].name);
                    strcat(opdu->data, "|");
                    strcat(content, list[i].name);
                }
            }
            sendto(s, opdu, sizeof(*opdu), 0, (const struct sockaddr *)&fsin,
sizeof(fsin));
        } else if (rpdu.type == 'T') {
            deregistration(s, rpdu.data, fsin);
        } else {
            printf("Error: Invalid type recieved\n");
        }
        printf("\n");
    }
    return 0;
}
```

```
void search(int s, char *data, struct sockaddr_in addr) {
    printf("Recieved search request with data: %s\n", data);
    char *sep = strtok(data, "|"); // Peer Name
    char peer[20];
    strcpy(peer, sep);
    // strcpy(list[max_index].name, contentName);

    char contentName[20]; // Content Name
    sep = strtok(NULL, "|");
    strcpy(contentName, sep);

    int i = 0;
    int found = 0;
    int found_index = 0;
    int minDownloads = 999;
    while (i < max_index) {
        if (!isEntryNULL(i) && strcmp(contentName, list[i].name) == 0 &&
list[i].head->downloads < minDownloads) {
            found_index = i;
            minDownloads = list[i].head->downloads;
            found = 1;
        }
        i++;
    }
    if (found == 1) {
        printf("Found registered content");
        PDU *spdu = malloc(sizeof(PDU));
        bzero(spdu->data, 100);
        spdu->type = 'S';
        strcat(spdu->data, list[found_index].name);
        strcat(spdu->data, "|");
        strcat(spdu->data, list[found_index].head->usr);
        strcat(spdu->data, "|");
        strcat(spdu->data, inet_ntoa(list[found_index].head->addr.sin_addr));
        strcat(spdu->data, "|");
        char port[10];
        sprintf(port, "%d", ntohs(list[found_index].head->addr.sin_port));
        strcat(spdu->data, port);
        printf("Search data to be returned to peer: %s \n", spdu->data);
        sendto(s, spdu, sizeof(*spdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));
        list[found_index].head->downloads += 1;
```

```
    } else {
        PDU *epdu = malloc(sizeof(PDU));
        bzero(epdu->data, 100);
        epdu->type = 'E';
        strcpy(epdu->data, "Content not found!");
        sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));
    }
    // Print contents of list
    void printEntry(ENTRY *entry) {
        printf("User: %s\n", entry->usr);
        printf("IP Address: %s\n", inet_ntoa(entry->addr.sin_addr));
        printf("Port: %d\n", ntohs(entry->addr.sin_port));
        printf("Downloads: %d\n", entry->downloads);
    }
    void printList(LIST *list) {
        printf("Content Name: %s\n", list->name);

        ENTRY *current = list->head;
        while (current != NULL) {
            printf("Entry:\n");
            printEntry(current);
            current = current->next;
        }
    }
    for (i = 0; i < max_index; i++) {
        printf("List %d:\n", i);
        printList(&list[i]);
        printf("\n");
    }
}

void deregistration(int s, char *data, struct sockaddr_in addr) {
printf("Data Recieved from deregistration: %s\n", data);
    char *sep = strtok(data, "|"); // Content Name
    char contentName[20];
    strcpy(contentName, sep);
    strcpy(list[max_index].name, contentName); //Might not need this

    char peer[20]; // Peer Name
    sep = strtok(NULL, "|");
    strcpy(peer, sep); //Variable for peer name, important
```

```
    int i = 0;
    int found = 0;


    while (!found && i < max_index) {
        if (!isEntryNULL(i) && strcmp(contentName, list[i].name) == 0 &&
strcmp(peer, list[i].head->usr) == 0) {
            printf("Found content to deregister: %s\n", contentName);
            // Found the content which is registered, need to remove it
        strcpy(list[i].name, ""); //This should replace that content with
NULL, therefore removing it
            found = 1;
        } else {
            printf("Content not registered %s and %s\n", contentName,
list[i].name);
        }
        i++;
    }
    if (found == 1) {
    //This is where we remove that filename from the list change from adding
to removing omehow?

        //max_index++; dont change the max index when removing, just remove
form the list just ignore for now

        PDU *apdu = malloc(sizeof(PDU));
        bzero(apdu->data, 100);
        apdu->type = 'A';
        strcpy(apdu->data, "File has been deregistered!\n");
        sendto(s, apdu, sizeof(*apdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));

    } else {

    //Final code if file is not registered
    PDU *epdu = malloc(sizeof(PDU));
        bzero(epdu->data, 100);
        epdu->type = 'E';
        strcpy(epdu->data, "File is not registered, therefore cannot be
removed");
```

```
        sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));
    }
}

void registration(int s, char *data, struct sockaddr_in addr) {
    printf("Data Recieved from Registration: %s\n", data);

    char *sep = strtok(data, "|"); // Content Name
    char contentName[20];
    strcpy(contentName, sep);
    strcpy(list[max_index].name, contentName);

    char peer[20]; // Peer Name
    sep = strtok(NULL, "|");
    strcpy(peer, sep);

    int i = 0;
    int found = 0;
    // Find out if there is existing content and peer already registered
    while (!found && i < max_index) {
        if (!isEntryNULL(i) && strcmp(contentName, list[i].name) == 0 &&
strcmp(peer, list[i].head->usr) == 0) {
            printf("Found existing content: %s", contentName);
            // Need to add new content server to content
            found = 1;
        } else {
            // No match found yet
            // printf("No match between %s and %s, %s, and %s\n", contentName,
list[i].name, peer, list[i].head->usr);
        }
        i++;
    }

    if (found == 1) {
        // Found an existing content and peer already registered, so sending
an error
        PDU *epdu = malloc(sizeof(PDU));
        bzero(epdu->data, 100);
        epdu->type = 'E';
        strcpy(epdu->data, "Peer and file already registered");
```

```
        sendto(s, epdu, sizeof(*epdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));
    } else {
        ENTRY *entry = malloc(sizeof(ENTRY));
        list[max_index].head = entry;
        list[max_index].head->downloads = 0;
        strcpy(entry->usr, peer);

        sep = strtok(NULL, "|"); // IP Address
        inet_aton(sep, (struct in_addr *)&entry->addr.sin_addr);

        int portNum;
        sep = strtok(NULL, "|");  // Port Number
        sscanf(sep, "%d", &portNum);
        entry->addr.sin_port = htons(portNum);

        printf("Port Number: %d\n", portNum);
        printf("Content Name: %s\n", list[max_index].name);
        printf("Peer Name: %s\n", entry->usr);
        printf("Port %d\n", ntohs(entry->addr.sin_port));

        max_index++;

        PDU *apdu = malloc(sizeof(PDU));
        bzero(apdu->data, 100);
        apdu->type = 'A';
        strcpy(apdu->data, "File has been registered!");
        sendto(s, apdu, sizeof(*apdu), 0, (const struct sockaddr *)&addr,
sizeof(addr));
    }
}

int isEntryNULL(int i) {
    // Make sure that there is no content available
    return (!strcmp(list[i].name, "") && list[i].head == NULL);
}
```

Peer:

```
/* A P2P client
It provides the following functions:
```

```
- Register the content file to the index server (R)
- Contact the index server to search for a content file (D)
        - Contact the peer to download the file
        - Register the content file to the index server
- De-register a content file (T)
- List the local registered content files (L)
- List the on-line registered content files (O)
*/

#include <arpa/inet.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define QUIT "quit"
#define BUFLEN 100        // Max buffer length
#define SERVER_PORT 14000 // Default server port
#define NAMESIZ 20        // Max size of Peer Name
#define MAXCON 200        // Max number of connections

typedef struct pdu
{
    char type;
    char data[BUFLEN];
} PDU;
PDU rpdu;

struct
{
    int sd;             // Socket Descriptor
```

```
    char name[NAMESIZ]; // Content Name
} table[MAXCON];        // Keep track of the registered content
int max_index = 0;      // Index for table

char usr[NAMESIZ];

int s_sock, peer_port;
int fd, nfds;
fd_set rfds, afds;

void registration(int, char *, struct sockaddr_in);
int search_content(int, char *, PDU *, struct sockaddr_in);
int client_download(char *, PDU *, struct sockaddr_in);
void server_download(int, struct sockaddr_in);
void deregistration(int, char *, struct sockaddr_in);
void online_list(int, struct sockaddr_in);
void local_list();
void quit(int, struct sockaddr_in);
void handler();

int main(int argc, char **argv)
{
    int server_port = SERVER_PORT;
    int n;
    struct sockaddr_in server;
    int alen = sizeof(struct sockaddr_in);
    struct hostent *hp;
    char c, *host, name[NAMESIZ];
    struct sigaction sa;

    switch (argc)
    {
    case 2:
        host = argv[1];
        break;
    case 3:
        host = argv[1];
        server_port = atoi(argv[2]);
        break;
    default:
        printf("usage: peer [host] [port]");
        exit(1);
```

```
}

// UDP connection with index server
memset(&server, 0, alen);
server.sin_family = AF_INET;
server.sin_port = htons(server_port);
if (hp = gethostbyname(host))
{
    memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
}
else if ((server.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
{
    printf("Can't get host entry\n");
    exit(1);
}

// Allocate a socket for the index server
s_sock = socket(PF_INET, SOCK_DGRAM, 0);
if (s_sock < 0)
{
    printf("Can't create a socket\n");
    exit(1);
}
if (connect(s_sock, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    printf("Can't connect\n");
    exit(1);
}

// Choose a username (used for Peer Name)
printf("Choose a username\n");
scanf("%s", usr);

nfds = 0;
// Initialize table structure to contain all connected peers
for (n = 0; n < MAXCON; n++)
{
    table[n].sd = -1;
}

// Setup signal handler
sa.sa_handler = handler;
```

```
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigaction(SIGINT, &sa, NULL);

    // Initialization for select()
    FD_ZERO(&afds);         // Clear fd_set
    FD_SET(0, &afds);       // Listening on read descriptor (stdin)
    FD_SET(s_sock, &afds); // Listening on server_socket

    int option;
    while (1)
    {
        printf("Click enter to do another action.\n");
        memcpy(&rfds, &afds, sizeof(rfds));
        if (select(FD_SETSIZE, &rfds, NULL, NULL, NULL) == -1)
        {
            printf("Select error: %s\n", strerror(errno));
            exit(1);
        }

        if (FD_ISSET(0, &rfds))
        { /* Command from the user  */
            printf("Select an option:\n");
            printf("(1) Register content\n");
            //printf("(2) List local content\n");
            printf("(2) List registered content\n");
            printf("(3) Download content\n");
            printf("(4) Deregister content\n");
            printf("(5) Quit\n");
            scanf("%d", &option);

            FILE *fp = NULL;
            char filename[10];

            switch (option)
            {
            case 1:
                printf("You chose Option 1.\n");

                while (fp == NULL)
                {
                    printf("Enter filename to register:\n");
```

```
                    scanf("%s", filename);
                    fp = fopen(filename, "rb");
                    if (fp == NULL)
                    {
                        printf("File does not exist, please select another
file\n");
                    }
                }

                registration(s_sock, filename, server);
                break;
            case 2:
                printf("You chose Option 2.\n");
                // call online_list() from index server
                online_list(s_sock, server);
                break;
            case 3:
                printf("You chose Option 3.\n");
                // call search_content()
                // call client_download()
                // call registration()
                printf("Enter file name to download:\n");
                scanf("%s", filename);
                PDU *rpdu = malloc(sizeof(PDU));
                bzero(rpdu->data, 100);
                int search_result = search_content(s_sock, filename, rpdu,
server);
                if (search_result == 1)
                {
                    printf("ERROR: File not found on index server, please use
a valid file name.\n");
                }
                else
                {
                    printf("File found on index server. Requesting
download.\n");
                    client_download(filename, rpdu, server);
                    registration(s_sock, filename, server);
                }

                break;
            case 4:
```

```
                printf("You chose Option 4.\n");
                while (fp == NULL)
                {
                    printf("Enter filename to de-register:\n");
                    scanf("%s", filename);
                    fp = fopen(filename, "rb");
                    if (fp == NULL)
                    {
                        printf("File does not exist, please select another
file\n");
                    }
                }
                deregistration(s_sock, filename, server);
                break;
            case 5:
                printf("You chose Option 5.\n");
                quit(s_sock, server);
                exit(0);
            default:
                printf("Invalid option. Please select a number between 1 and 6
inclusive.\n");
            }
        }
        else
        {
            // Download was requested, send file to content client from
content server
            printf("A download was requested by another peer, addressing
download.\n");
            server_download(s_sock, server);
        }
        printf("\n");
    }
}

void quit(int s_sock, struct sockaddr_in server)
{
char dereg[10];
    printf("Quitting server, deregistering all content...\n");
    int i;
    for (i = 0; i < max_index; i++){
        strcpy(dereg, table[i].name);
```

```
        printf("Degresitering: %s\n", dereg);
        deregistration(s_sock, dereg, server);
    }
    printf("Done\n");
    max_index = 0;
    exit(0);
}


void online_list(int s_sock, struct sockaddr_in server)
{
    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'O';
    strcat(rpdu->data, "send list please");

    // printf("Registering peer %s \n", usr);
    // printf("Data sent for online list: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server,
sizeof(server));

    // Recieve acknowledgement of content registration
    PDU recv_pdu;
    int i, n, len;
    char listed_files[100];
    char *token;
    int index=0;
    n=recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server,
&len);
         if (recv_pdu.type == 'O'){
            printf("Received registered files from index server: \n");
            token = strtok(recv_pdu.data, "|");
            while (token != 0)
            {
                printf("[%d] %s \n", index, token);


                token = strtok(0, "|");
                index++;
            }
       }else{
            printf("error listing \n");
```

```
        }

}

void server_download(int s_sock, struct sockaddr_in server)
{
    struct sockaddr_in client;
    int descriptor;
    int current_sd;
    int n;
    FILE *file_pointer_server;
    char fname[100];

    for (descriptor = 0; descriptor < MAXCON; descriptor++)
    {

        if (FD_ISSET(table[descriptor].sd, &rfds))
        {
            current_sd = table[descriptor].sd;
            break;
        }
    }
    int client_len = sizeof(client);
    int new_sd = accept(current_sd, (struct sockaddr *)&client, &client_len);

    PDU *recv=malloc(sizeof(PDU)) ; // setting up send pdu to content server
    bzero(recv->data, 100);
    char *token;
    int index = 0;
    // gets the file name to be downloaded
    //while (n = read(new_sd, recv, sizeof(PDU)) > 0)
    //{
        n = read(new_sd, recv, sizeof(PDU));
        printf("bytes received %d\n",n);
        printf("CONNECTION TO SERVER DOWNLOAD SUCCESS! \n");
        printf("type received is %c \n",recv->type);
        printf("data is %s\n",recv->data);
        if (recv->type == 'D')
        {
            token = strtok(recv->data, "|");
            while (token != 0)
            {
```

```
                if (index == 1)
                {
                    strcpy(fname, token);
                }

                token = strtok(0, "|");
                index++;
            }
        }

    printf("Name of file to open is: %s \n",fname);
    file_pointer_server = fopen(fname, "r");
    PDU resp;
    if (file_pointer_server == NULL)
    {
        printf("file failed to open");
        resp.type = 'E';
        // resp.data = "FILE NOT FOUND";
        strcpy(resp.data, "FILE NOT FOUND");
        write(new_sd, &resp, sizeof(resp));
    }
    else
    {
        while (fgets(resp.data, 100, file_pointer_server) != NULL)
        {
            resp.type = 'C';
            write(new_sd, &resp, sizeof(resp));
        }
        fclose(file_pointer_server);
    }
    (void)close(new_sd);
}

int search_content(int s_sock, char *name, PDU *rpdu, struct sockaddr_in
server)
{
    /* Checks if index server has the file, that is to be downloaded
    -communicates to the index server via udp
    -returns 1 if file not found , returns 0 if success
    */

    PDU *spdu = malloc(sizeof(PDU));
```

```
    bzero(spdu->data, 100);

    spdu->type = 'S';
    // strcat(spdu->type, 'S'); // type
    strcat(spdu->data, usr); // peer name
    strcat(spdu->data, "|");
    strcat(spdu->data, name); // content name
    printf("Search data sent to index server: %s\n", spdu->data);
    printf("type is: %c \n", spdu->type);
    sendto(s_sock, spdu, sizeof(*spdu), 0, (const struct sockaddr *)&server,
sizeof(server));
    int len;
    PDU *recv_pdu;
    int alen = sizeof(server);
    recvfrom(s_sock, rpdu, sizeof(PDU), 0, (struct sockaddr *)&server, &alen);
    printf("Search data received from index server: %s\n", rpdu->data);

    if (rpdu->type == 'E')
    {
        printf("Error from search request: %s\n", rpdu->data);
        return 1;
    }
    else
    {
        // type S
        printf("File found! \n");
        printf("Data Received: %s \n", rpdu->data);
        return 0;
    }
}

int client_download(char *name, PDU *pdu, struct sockaddr_in server)
{
    /* Communicates with a content Server
       -connects to content server tcp socket and initiates download.
       -returns 1 if fail, returns 0 if success
   */
    // create file using name , name is content name
    struct sockaddr_in content_server;
    int sd;
    char address[100];
    strcpy(address, pdu->data);
```

```
    int index = 0;
    int n;
    char *ip;
    int port;
    PDU spdu; //= malloc(sizeof(PDU)); // setting up send pdu to content
server
    bzero(spdu.data, 100);
    char t='D';
    spdu.type='D';
    strcpy(spdu.data, usr); // follows format of ->  D | peer name | content
name
    strcat(spdu.data, "|");
    strcat(spdu.data, name);
    // getting ip and port of content server
    char *token;
    token = strtok(address, "|");

    while (token != 0)
    {
        if (index == 2)
        {
            //strcpy(ip, token);
            ip=(char *)token;
        }
        if (index == 3)
        {
            port = atoi(token);
        }
        token = strtok(0, "|");
        index++;
    }
    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }
    // setup connection
    printf("Setting up connection with content server\n");

    bzero((char *)&content_server, sizeof(struct sockaddr_in));
    content_server.sin_family = AF_INET;
    content_server.sin_port = htons(port);
```

```
   // strcpy((char *)&content_server.sin_addr, ip);
    int inet_t=inet_aton(ip,(struct in_addr *)&content_server.sin_addr);
    printf("result of inet %d \n",inet_t);
    if(inet_t==0){
        printf("error\n");
    };
    if (connect(sd, (struct sockaddr *)&content_server,
sizeof(content_server)) == -1)
    {
        fprintf(stderr, "Can't connect \n");
        exit(1);
    }
    printf("Sending download request to content server with data: %s\n",
spdu.data);
    printf("with type %c \n",spdu.type);
    write(sd, &spdu, sizeof(PDU));
    // content server replies with C type
    PDU rpdu;
    FILE *file_pointer;
    file_pointer = fopen(name, "w");
    while (n = read(sd, &rpdu, sizeof(struct pdu)) > 0)
    {
        if (rpdu.type == 'C')
        {
            fputs(rpdu.data, file_pointer);
        }
        else if (rpdu.type == 'E')
        {
            printf("ERROR FILE DOWNLOAD FAILED \n");

            fclose(file_pointer);
            remove(name);
            return 1;
        }
    }

    fclose(file_pointer);

    //registration(s_sock, name, server);

    return 0;
}
```

```
void deregistration(int s_sock, char *name, struct sockaddr_in server)
{

    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'T'; //De register a file name
    strcat(rpdu->data, name);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, usr);
    strcat(rpdu->data, "|");
   // strcat(rpdu->data, inet_ntoa(reg_addr.sin_addr));
    //strcat(rpdu->data, "|");
    //strcat(rpdu->data, port);
    printf("Deregistering peer %s \n",usr);
    printf("Data Sent: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server,
sizeof(server));


 // Recieve acknowledgement of content deregistration
    PDU recv_pdu;
    int i, n, len, s;
    recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server,
&len);

    if (recv_pdu.type == 'A')
    {
    printf("File has been deregistered %s \n",name);
        strcpy(table[max_index].name, name);
    int i;
        for (i = 0; i < max_index; i++)
        {
            if (strcmp(table[i].name, name) && s >= nfds) //This finds the
filename from the struct and removes it
            {
        strcpy(table[i].name, ""); //This removes file from struct
                nfds = s + 1;
        table[i].sd = s;
        FD_CLR(s, &afds); //This clears socket
        table[i].sd = -1;           //Remove socket from struct
        close(s);               //Close TCP connection
```

```
            }
        }

    }
    else
    {
        printf("Error : %s\n", recv_pdu.data);
    }
}

void registration(int s_sock, char *name, struct sockaddr_in server)
{
    /* Register a content to the index server
        - Create a TCP socket for other peers to download
        - Add TCP socket descriptor and content name to local list
        - Register the content to the index server via UDP
        - Update nfds (number of file descriptors)
    */
    int s;
    // Create a stream socket
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    { // Create TCP socket for content
        fprintf(stderr, "Can't create a socket\n");
        exit(1);
    }

    // Bind an address to the socket
    struct sockaddr_in reg_addr; // TCP Socket Address
    bzero((char *)&reg_addr, sizeof(struct sockaddr_in));
    reg_addr.sin_family = AF_INET;
    reg_addr.sin_port = htons(0); // Is 0 correct?
    reg_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind(s, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) == -1)
    {
        fprintf(stderr, "Can't bind name to socket\n");
        exit(1);
    }

    // Queue upto max 5 connect requests
    listen(s, 5);

    char port[10];
```

```
    int alen = sizeof(struct sockaddr_in);
    getsockname(s, (struct sockaddr *)&reg_addr, &alen); // Get socket address
information
    sprintf(port, "%d", ntohs(reg_addr.sin_port));       // Get port of socket
and put into string

    PDU *rpdu = malloc(sizeof(PDU));
    bzero(rpdu->data, 100);
    rpdu->type = 'R';
    strcat(rpdu->data, name);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, usr);
    strcat(rpdu->data, "|");
    strcat(rpdu->data, inet_ntoa(reg_addr.sin_addr));
    strcat(rpdu->data, "|");
    strcat(rpdu->data, port);
    // printf("Registering peer %s \n", usr);
    printf("Registration data sent to index server: %s\n", rpdu->data);
    sendto(s_sock, rpdu, sizeof(*rpdu), 0, (const struct sockaddr *)&server,
sizeof(server));

    // Recieve acknowledgement of content registration
    PDU recv_pdu;
    int i, n, len;
    recvfrom(s_sock, &recv_pdu, sizeof(PDU), 0, (struct sockaddr *)&server,
&len);

    if (recv_pdu.type == 'A')
    {
        strcpy(table[max_index].name, name);
    int i;
        for (i = 0; i < max_index; i++)
        {
            if (strcmp(table[i].name, "") && s >= nfds)
            {
                nfds = s + 1;
            }
        }
        table[max_index].sd = s;
        FD_SET(s, &afds);
        max_index++;
    }
```

```
    else
    {
        printf("Error: %s\n", recv_pdu.data);
    }
}

void handler()
{
    //quit(s_sock);
}
```