Toronto Metropolitan University

Department of Electrical, Computer and Biomedical Engineering

Course Number	COE768
Course Title	Computer Networks
Semester/Year	Fall 2023
Instructor	Truman Yang

Lab Report #	1
Lab Title	COE768 Project Report
Date	2023-11-30
Section	04
Section TA	Parastoo Jafarzadeh

Student Name	Student ID	Signature
Edwin Chen	500959809	EC
Mohib Khan	500953221	MK
Thomas Pazhaidam	500952854	TP

*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: www.ryerson.ca/senate/current/pol60.pdf

Introduction:

In this project, we are tasked to develop a Peer-to-Peer (P2P) application that consists of an Index server and multiple peers. With the support of the Index server, the peers should be able to exchange content among themselves. Peers should be able to act as both content servers, making their content available for download, and as content clients, seeking content from other peers. The content server should register its content with the index server, and the content client should retrieve the address of a content server from the index server. The communication between the index server and peers is based on UDP, while content download occurs over TCP.

Socket programming in C for Linux is very important for constructing applications that allow for communication between devices over a network. Sockets, which act as communication endpoints, can allow for processes on different machines to establish connections and share data between each other. You can use the C language and system calls to create, bind, and manage sockets. There are two main socket types that are used, Stream (TCP) which offers reliable, connection-oriented, and Datagram (UDP) which is connectionless, unreliable communication.

Description of the Peer and Index program:

<u>UDP Socket Connection</u>

```
s = socket(AF_INET, SOCK_DGRAM, 0);
if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
             fprintf(stderr, "Can't connect to %s \n", host);
```

Figure 1.1: initializing peer UDP connection

Implementing the UDP protocol, for the Peer, the UDP socket is created using the 'socket()' function with 'SOCK_DGRAM' flag, indicating that it will be used for datagram-oriented communication. The 'connect()' function is then employed to establish a connection to the server, effectively binding the UDP socket to the server's address and port. This connection is essential for subsequent 'write()' and 'read()' (as shown in figure 1.2) operations to interact with the index.

```
    write(Socket, &msg, sizeof(msg));
    struct PDU resp;
    read(Socket, &resp, sizeof(msg));
```

Figure 1.2: Peer UDP read and write functions

It is important to remember that UDP is connectionless so the connect function just stores the port and IP of the index. The code utilizes UDP for specific operations, such as querying the list of online content. In this function, a UDP message (PDU) is constructed, with the 'msg.type' set to 'ONLINE', indicating the request for online content information. The constructed message is then sent to the index using the 'write()' function, and the index's response is received through the 'read()' function.

```
        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);
```

Figure 1.3: UDP index socket initialization

On the Index side, the code initializes a UDP socket using the 'socket()' function with the 'SOCK_DGRAM' flag. The server binds to a specific port using the 'bind()' function. The main loop listens for incoming UDP messages from peers using 'recvfrom()' (as shown in figure 1.4), which receives messages into a 'buffer (socket_buf)'.

```
                if (recvfrom(s, socket_buf, sizeof(socket_buf), 0, (struct sockaddr *)&fsin,
&alen) < 0) {
                        printf("recvfrom error\n");
                }
sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr *)&fsin, sizeof(fsin));
```

Figure 1.4: rx and tx functions for index

The received data is then copied into a struct'(struct PDU)', facilitating the extraction of the message type and data payload. Depending on the message type, different operations are performed, such as content registration, content deregistration, content search, content listing and program quitting. Responses are then sent back to the peers using the 'sendto()' (as shown in figure 1.4) function.

TCP Socket Connection

```
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));

         struct ifreq ifr;
         char array[] = "enp0s3";

        reg_addr.sin_family = AF_INET;
         strncpy(ifr.ifr_name , array , IFNAMSIZ - 1);
         ioctl(sd, SIOCGIFADDR, &ifr);
        reg_addr.sin_port = htons(0);
        reg_addr.sin_addr.s_addr = inet_addr(inet_ntoa(( (struct sockaddr_in *)&ifr.ifr_addr
)->sin_addr));
         fflush(stdout);
         printf("%s\n", inet_ntoa(( (struct sockaddr_in *)&ifr.ifr_addr )->sin_addr));
         fflush(stdout);
        if (bind(sd,(struct sockaddr*)&reg_addr, sizeof(reg_addr)) == -1){
                fprintf(stderr, "Can't bind name to socket\n");
                exit(EXIT_FAILURE);
        }
```

Figure 1.5: tcp server socket initialization

Implementing the TCP protocol, for the Peer, The IP of the peer is dynamically retrieved from the network adapter and is then bound to the server side socket, the peer then just waits for a connection request when a download is requested and acts accordingly.
The code also involves setting up a TCP client for downloading files from another peer in InitializeTCPDownload() (figure 1.6).

```
if ((tcp_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
fprintf(stderr, "Can't create a socket\n");
exit(1);
}

bzero((char*)&server, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
int port = atoi(Port);

server.sin_port = port;
printf("%d", server.sin_port);
if (host = gethostbyname(Address)){
bcopy(host->h_addr, (char *)&server.sin_addr, host->h_length);
}
else if (inet_aton(Address, (struct in_addr *) &server.sin_addr)){
fprintf(stderr, "Can't get server's address\n");
}

if (connect(tcp_socket, (struct sockaddr *)&server, sizeof(server)) == -1){
fprintf(stderr, "cant connect to server: %s\n", host->h_name);
}
```

Figure 1.6: tcp client socket connection request

<u>Select function for incoming requests</u>

In the main function of the peer a select function is used to read incoming transaction requests from the stdin and tcp client connections. Based on the type of transaction either the incoming client connection will be established for a file download and the file from the peer content server will be transferred to the peer content client via handle_client() function. Afterwards the tcp connection will be closed using the close() command. Otherwise, the transaction will be considered to be an output in the terminal and one of the options will be selected as requested by the peers user.

```
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);
        }
}
if(FD_ISSET(fileno(stdin), &rfds)) {
        char input[10] = {0};
```
Figure 1.7: select function for connection request/user input

<u>Register Content</u>

Upon initiation of the program, the peer registers its content with the network using the 'RegisterContent()' function. This involves gathering information about the content, such as its name, description, and file path, and packaging it into a registration request.

```
    char data[BUFFLEN] = {0};
    strcpy(data, PeerName);
    strcat(data, ";");
    strcat(data, Filename);
    strcat(data, ";");
    strcat(data, tcpAddr);
    strcat(data, ";");
    strcat(data, tcpPort);
    strcpy(msg.data, data);
    printf("Here is the data: %s\n", msg.data);

    write(UdpSocket, &msg, sizeof(msg));
    fflush(stdout);
    read(UdpSocket, Response, sizeof(Response));
```
Figure 1.8: send details of file to register to index

The registration request is sent to the "Index" server, which updates its database to include the newly registered content.

```
strcpy(registered_content[i].content, content);
strcpy(registered_content[i].peer, peer);
strcpy(registered_content[i].address, addr);
strcpy(registered_content[i].port_number, port);
```
Figure 1.9: store registered content information into global array

For the index, upon receiving a registration request, the 'RegisterContent()' function from the Peer updates the internal database and store the information in the 'registered_content[]' array, with information about the newly registered content, including details like name, description, and the peer hosting the content (as shown in figure 1.9).

<u>Search Content</u>

Content discovery is facilitated by the 'SearchContent()' and RequestOnlineContent() functions which are executed when the user uses their corresponding commands. When a user queries the network for specific content, the peer sends a search request to the server. The server responds with a list of relevant content, enabling users to discover and download files from other peers.

Download Content

```
while (1) {
            n = recv(tcpSocket, &resp, PACKETSIZE, 0);
            resp.data[BUFFLEN] = '\0';
            if (resp.type == ERR){
                   printf("ERROR: %s\n", resp.data);
                   remove(contentName);
                   return -1;
            }
            fprintf(file, "%s", resp.data);      // Write to file
```

Figure 1.9: download content to peer requesting file

Downloading content is a more involved process handled by the 'DownloadContent()' function. This function utilizes the TCP protocol for reliable data transfer. Upon selecting a specific content to download, the peer establishes a TCP connection with the peer hosting the desired content.

Deregister Content

```
msg.type = (int) DEREGISTER;
strcpy(msg.data, SelfPeerName);
strcat(msg.data, ";");
strcat(msg.data, content);

write(Socket, &msg, sizeof(msg));
```

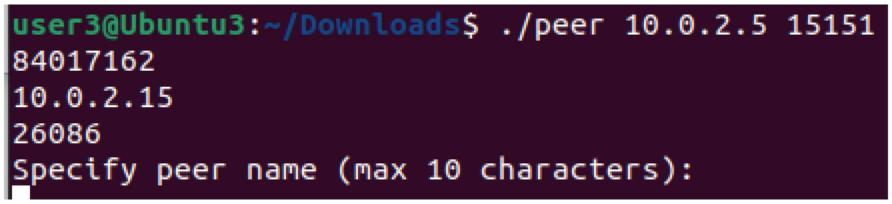
Figure 1.9: deregister request to index from peer

In scenarios where a peer decides to remove its content from the network, the 'DeregisterContent()' function comes into play. This function sends a deregistration request to the "Index" server over UDP, prompting the server to update its records and indicate that the specified content is no longer available for sharing.

Observations & Analysis:

Peer Initialization:

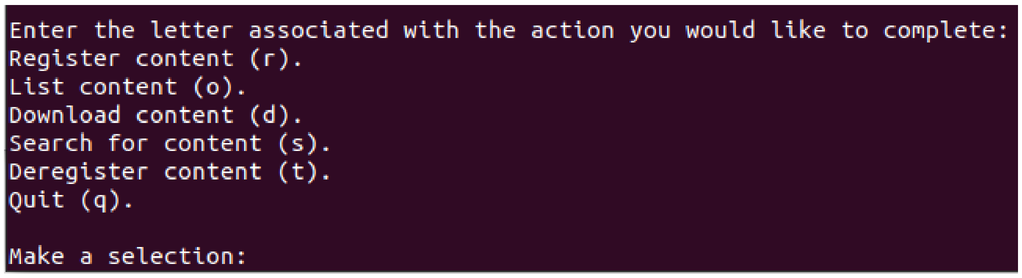
The peer user will need to input a peer name to be registered to the server.



```
user3@Ubuntu3:~/Downloads$ ./peer 10.0.2.5 15151
84017162
10.0.2.15
26086
Specify peer name (max 10 characters):
```

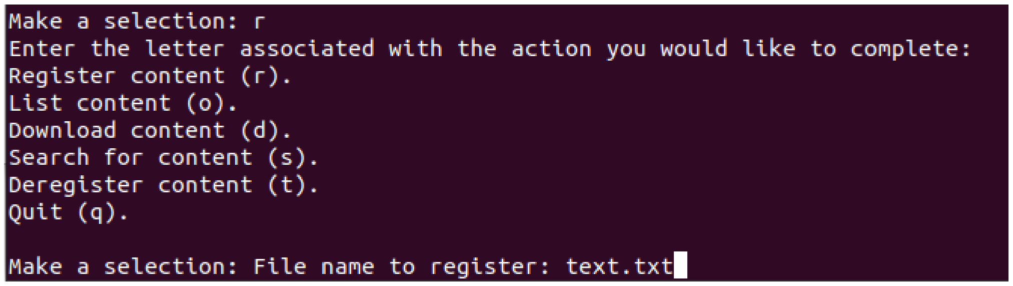
Main Menu:

The peer user will choose which action they would like to take and input the corresponding letter.



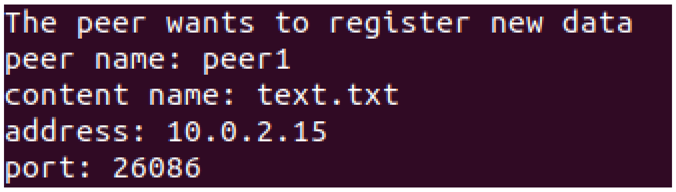
```
Enter the letter associated with the action you would like to complete:
Register content (r).
List content (o).
Download content (d).
Search for content (s).
Deregister content (t).
Quit (q).

Make a selection:
```

Register Content (peer):
If the peer inputs ‘r’, they will then be prompted to enter the name of the file that will be registered.



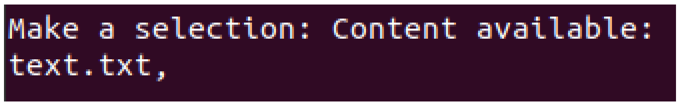
```
Make a selection: r
Enter the letter associated with the action you would like to complete:
Register content (r).
List content (o).
Download content (d).
Search for content (s).
Deregister content (t).
Quit (q).

Make a selection: File name to register: text.txt
```

Index Terminal:
The following information will be shown in the terminal of the index.



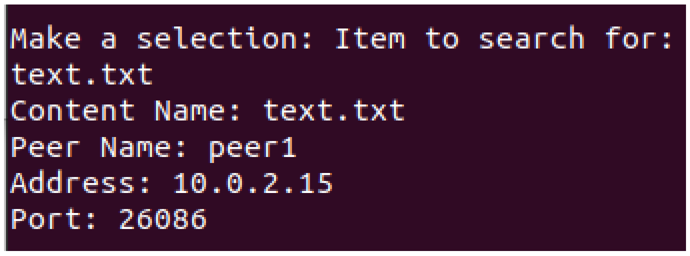
```
The peer wants to register new data
peer name: peer1
content name: text.txt
address: 10.0.2.15
port: 26086
```

List of all available content on the server:
If ‘o’ was inputted, then a list of all available content currently registered on the index.



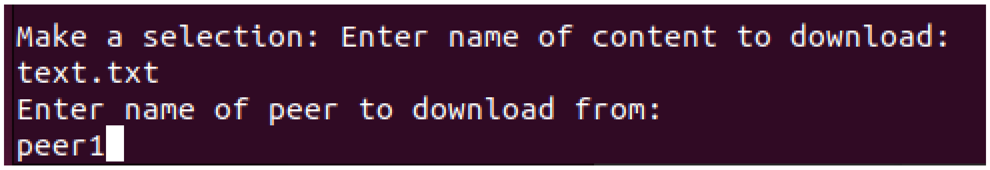
```
Make a selection: Content available:
text.txt,
```

Searches for item registered by peers:
If ‘s’ was inputted, then the peer will be prompted to enter the name of the file that they want to search for and what will be shown is the information about the file, such as the peer name, IP address and Port number.



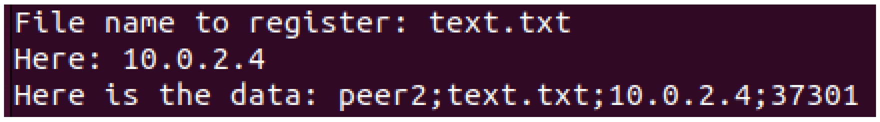
```
Make a selection: Item to search for:
text.txt
Content Name: text.txt
Peer Name: peer1
Address: 10.0.2.15
Port: 26086
```

Download request from a different peer:
If ‘d’ was inputted, the peer will be prompted to enter the name of the file that they want to download. Next they will be prompted to enter which peer they want to download from.



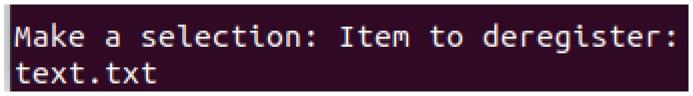
```
Make a selection: Enter name of content to download:
text.txt
Enter name of peer to download from:
peer1
```

Request to register recently downloaded file:
After the peer has downloaded the file, they will be prompted to register the file that they recently downloaded.



```
File name to register: text.txt
Here: 10.0.2.4
Here is the data: peer2;text.txt;10.0.2.4;37301
```

Deregister content from index:
If ‘t’ is inputted, the peer will be prompted to enter the name of the file that they want to deregister from the index.



```
Make a selection: Item to deregister:
text.txt
```

Quit program:
If ‘q’ was inputted, the program will end and quit to the terminal.

Conclusions:
In conclusion, creating a Peer-to-Peer (P2P) application with an Index server and multiple peers provides a strong solution for sharing content in a decentralized way. Using socket programming in C for Linux, the system effectively uses UDP for quick communication between the Index server and peers, and TCP for dependable content downloads. Peers can act as both content servers and clients, thanks to their interaction with the Index server, making the system more versatile. This project highlights the importance of socket programming in enabling smooth communication between devices on a network, with TCP and UDP serving different but complementary roles in making a dynamic and efficient P2P application.

Appendix:
Source Code for Index

```
// COE768 - Design Project
// November 12th, 2023
// Mohib Khan, Edwin Chen, Thomas Pazhaidam
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/unistd.h>
#include <sys/signal.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <netdb.h>
#include <stdio.h>
#include <time.h>

// Stucture of the PDU
struct PDU {
      char type;
      char data[100];
};

// Structure to store registered content
typedef struct{
      char content[10];
      char peer[10];
      char address[15];
      char port_number[10];
} reg_data;

// Definitions
#define BUFSIZE sizeof(struct PDU)
#define ERROR    'e' //report error
#define ONLINE   'o'
#define QUIT     'q'
#define REGISTER 'r'
#define DEREG    't'
#define SEARCH   's'
#define ACKNOWLEDGEMENT      'a'

// Main section
/////////////////////////////////////////////////////////////////////////////
/////////////
int main(int argc, char *argv[])
{
      // Variables
      struct  sockaddr_in fsin;
      char  buf[100];
      char    *pts;
      int   sock;
      time_t      now;
      int   alen;
      struct  sockaddr_in sin;
      int     s, type, error_found =0;
      int   port=3000;

      struct PDU msg, send_msg;
      int   socket_read_size, i=0;
      char  socket_buf[BUFSIZE];
      char  file_name[100], online_content[100];
      int   file_size, registered=0, found = 0, placement = 0;
      reg_data registered_content[10];
      char data_recieved[100], peer_name[10], content_name[10],
address[13], port_num[10];
      int content_tracking[] = {0,0,0,0,0,0,0,0,0,0};

      // Set up UDP server connection
      switch(argc){
            case 1:
                  break;
            case 2:
                  port = atoi(argv[1]);
                  break;
            default:
                  fprintf(stderr, "Usage: %s [port]\n", argv[0]);
                  exit(1);
      }

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

    /* Allocate a socket */
      s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0) {
            fprintf(stderr, "can't creat socket\n");
      }

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

      listen(s, 5);
      alen = sizeof(fsin);
      // Main section, after UDP connection is set up
      while (1) {

            // Reset variables
            memset(&send_msg, '\0', sizeof(send_msg));
            memset(&data_recieved, '\0', sizeof(data_recieved));
            registered = 0;
            found = 0;
            error_found = 0;

            // Get message from peer
            if (recvfrom(s, socket_buf, sizeof(socket_buf), 0, (struct
sockaddr *)&fsin, &alen) < 0) {
                  printf("recvfrom error\n");
            }
            // Copy message into PDU format
            memcpy(&msg, socket_buf, sizeof(socket_buf));
            printf("%s\n", msg.data);
            // Handle request for list of available content
            if (msg.type == ONLINE) {
                  // Get and parse message from peer
                  printf("Online content request\n");
                  memcpy(data_recieved, msg.data, sizeof(socket_buf) - 1);
                  // Prepare message to send to peer
                  send_msg.type = ONLINE;
                  for (i=0; i<10; ++i){ //Get list of existing content,
seperate by commas
                        if (content_tracking[i]==1){
                              strcat(send_msg.data,
registered_content[i].content);
                              strcat(send_msg.data, ", ");
                        }
                  }
                  // Send message to peer
                  sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));
            }
            // Handle request to deregister content
            if (msg.type == DEREG) {
                  // Get and parse message from peer
                  printf("Deregister content request\n");
                  memcpy(data_recieved, msg.data, BUFSIZE - 1);
                  //sscanf(data_recieved, "%[a-zA-Z0-9];%[a-zA-Z0-9]",
peer_name, content_name);
                  char *peer_name = strtok(data_recieved, ";");
                  char *content_name = strtok(NULL, ";");
```

```
                    // Print the peer and content name they want to
deregister
                    printf("peer name: %s\n", peer_name);
                    printf("content name: %s\n", content_name);

                    //Check and see if the content exists, and where it's
stored
                    for (i=0; i<10; ++i){
                          if ((strcmp(registered_content[i].content,
content_name)==0) && (content_tracking[i]==1) &&
(strcmp(registered_content[i].peer, peer_name)==0)){
                                placement = i;
                                found = 1;
                          }
                    }
                    // If content doesn't exist, send back an error
                    if (found == 0){
                          send_msg.type = ERROR;
                          strcpy(send_msg.data, "This content is not
registered with this peer");
                    }
                    // If content exists, remove it from storage, and send
back an acknowledgement
                    else if (found == 1){
                          //memset(&registered_content[placement], '\0',
sizeof(msg));
                          content_tracking[placement] = 0;
                          send_msg.type = ACKNOWLEDGEMENT;
                          strcpy(send_msg.data, "content has been
deregistered");
                    }
                    // Send message to peer
                    sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));
              }
              // Handle request to seach for content
              else if (msg.type == SEARCH){
                    // Get and parse message from peer
                    printf("Search content request\n");
                    memcpy(data_recieved, msg.data, BUFSIZE - 1);
                    char *peer = strtok(data_recieved, ";");
                    char *content = strtok(NULL, ";");
                    char *addr = strtok(NULL, ";");
                    char *port = strtok(NULL, ";");
                    // Print the peer and content name they want to find
                    printf("peer name: %s\n", peer);
                    printf("content name: %s\n", content);
```

```
                // Check and see if the content name exists in storage
                for (i=0; i<10; ++i){
                     if ((strcmp(registered_content[i].content,
content)==0) && (content_tracking[i]==1)){
                          placement = i;
                          found = 1;
                     }
                }
                // If the content doesn't exist, send back an error
                if (found == 0){
                     send_msg.type = ERROR;
                     strcpy(send_msg.data, "No content by this name and
peer was found");
                }
                //If the content exists, send back an 'S' type message
with the address and port number
                else if (found == 1){
                     send_msg.type = SEARCH;
                     printf("%s\n",
registered_content[placement].content);
                     strcpy(send_msg.data,
registered_content[placement].content);
                     strcat(send_msg.data, ";");
                     printf("%s\n", registered_content[placement].peer);
                     strcat(send_msg.data,
registered_content[placement].peer);
                     strcat(send_msg.data, ";");
                     printf("%s\n",
registered_content[placement].address);
                     strcat(send_msg.data,
registered_content[placement].address);
                     strcat(send_msg.data, ";");
                     printf("%s\n",
registered_content[placement].port_number);
                     strcat(send_msg.data,
registered_content[placement].port_number);
                }
                // Send message to peer
                sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));
           }
           // Handle request to quit the program
           else if (msg.type == QUIT){
                // Get and parse the message
                printf("The peer wants to quit the program\n");
                memcpy(peer_name, msg.data, BUFSIZE - 1);
                //Check to see if they have any open content
                for (i=0; i<10; ++i){
                     if ((strcmp(registered_content[i].peer,
peer_name)==0) && content_tracking[i]==1){
                          found = 1;
                     }
                }
                // If they still have open content, send back an error
                if (found == 1){
                     send_msg.type = ERROR;
                     strcpy(send_msg.data, "unable to close, content
still registered to this peer");
                     sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));
                }
                // If all their content is deregistered, send back 'Q'
type message
                else if (found == 0){
                     send_msg.type = QUIT;
                     sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));
                }
           }
           // Handle request to register content
           else if (msg.type == REGISTER){
                // Get and parse message
                printf("The peer wants to register new data\n");
                memcpy(data_recieved, msg.data, BUFSIZE - 1);
                char *peer = strtok(data_recieved, ";");
                char *content = strtok(NULL, ";");
                char *addr = strtok(NULL, ";");
                char *port = strtok(NULL, ";");
                //Print information recieved
                printf("peer name: %s\n", peer);
                printf("content name: %s\n", content);
                printf("address: %s\n", addr);
                printf("port: %s\n", port);
                //Find an empty spot to store the content information
                for (i=0; i<10; ++i){
                     if ((strcmp(registered_content[i].peer, peer)==0)
&& (strcmp(registered_content[i].content, content)==0) &&
content_tracking[i]==1){
                          error_found = 1;
                     }
                }
                if (error_found ==0){
                     for (i=0; i<10; ++i){
```

```
                              if ((content_tracking[i]==0) && (registered
==0)){
                                    strcpy(registered_content[i].content,
content);
                                    strcpy(registered_content[i].peer,
peer);
                                    strcpy(registered_content[i].address,
addr);

strcpy(registered_content[i].port_number, port);
                                    printf("%s\n",
registered_content[i].address);
                                    registered = 1;
                                    content_tracking[i] = 1;
                              }
                        }
                  }
                  //Send an acknowledgement if the content was stored
properly
                  if (registered == 1){
                        send_msg.type = ACKNOWLEDGEMENT;
                        strcpy(send_msg.data, "content has been
registered");
                  }
                  //Send an error if the content couldn't be stored
                  else if (registered == 0){
                        send_msg.type = ERROR;
                        strcpy(send_msg.data, "error registering content");
                  }
                  //Send message to the peer
                  sendto(s, &send_msg, BUFSIZE, 0, (struct sockaddr
*)&fsin, sizeof(fsin));

            }
      }
      // Exit the program
      close(s);
      exit(0);
}
```

Source Code for Peer

```
/***************************************************************************
******
* Title                 :   P2P Project
* Filename              :   peer.c
* Author                :   Thomas Pazhaidam, Mohib Khan, Edwin Chen
* Origin Date           :   11/11/2023
* Version                :   0.0.1
* Notes                  :   None
***************************************************************************
*****/
/** \file peer.c
 * \brief This module contains functionality of peers for a peer to peer
application
 */
/**************************************************************************
*****
* Includes
***************************************************************************
*****/
#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 <netdb.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <errno.h>

/**************************************************************************
*****
* Module Preprocessor Macros
***************************************************************************
*****/
#define BUFFLEN 100
#define PACKETSIZE 101
#define    NAMESIZE   10
/**************************************************************************
*****
* Module Typedefs
***************************************************************************
*****/
struct PDU {
    char type;
    char data[BUFFLEN];
};

struct pdu {
      char type;
      char peerName[NAMESIZE];
      char contentName[NAMESIZE];
      struct     sockaddr_in addr;
 };
/*******************************************************************************
*****
* Module Variable Definitions
*******************************************************************************
*****/
enum MsgType {
    REGISTER = 114, DOWNLOAD = 100, SEARCH = 115, DEREGISTER = 116, DATA =
99,
    ONLINE = 111, ACK = 97, ERR = 101, QUIT = 113
};
/*******************************************************************************
*****
* Function Prototypes
*******************************************************************************
*****/

/*******************************************************************************
*****
* Function Definitions
*******************************************************************************
*****/
/*******************************************************************************
*****
* Request user to specify peer name from terminal
*******************************************************************************
*****/
void GetPeerName(char *PeerName)
{
    int outLen = 0;
    //memset(&PeerName, 0, sizeof(PeerName));
    printf("Specify peer name (max 10 characters): \n");
    outLen = read(0, PeerName, 10);
    PeerName[outLen-1] = 0;
}

/*******************************************************************************
*****
* Request list of online content on server
*******************************************************************************
*****/
void RequestOnlineContent(int Socket)
{
    struct PDU msg, resp;
    memset(&msg, 0, sizeof(msg));
    memset(&resp, 0, sizeof(resp));
    msg.type = ONLINE;

    write(Socket, &msg, sizeof(msg));
    read(Socket, &resp, sizeof(resp));

    printf("Content available: \n");
    printf("%s\n", resp.data);
}

/**************************************************************************
*****
* Initialize TCP Connection and returns tcp server socket and server
response
***************************************************************************
*****/
int InitializeTCPUpload(int UdpSocket, char* PeerName, char* Filename,
struct PDU *Response, struct sockaddr_in reg_addr)
{
      //printf("function start");
      fflush(stdout);
      // Make tcp connection
      int   tcp_socket, alen;
      struct      sockaddr_in server;
    struct  in_addr ip_addr_struct;
      struct      PDU msg;
      char  tcpAddr[15], tcpPort[6], send_data[100];
      //printf("socket start");
    if ((tcp_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    fprintf(stderr, "Can't create a socket\n");
    exit(1);
      }
    bzero((char*)&server, sizeof(struct sockaddr_in));
    server.sin_family = AF_INET;
    // server.sin_port = htons(0);
    // server.sin_addr.s_addr = htonl(INADDR_ANY);

    server.sin_port = reg_addr.sin_port;
    server.sin_addr.s_addr = reg_addr.sin_addr.s_addr;

    //get current ip
    // struct ifreq ifr;
    // char array[] = "enp0s3";
    // strncpy(ifr.ifr_name , array , IFNAMSIZ - 1);
    // ioctl(tcp_socket, SIOCGIFADDR, &ifr);
    // if(bind(tcp_socket,(struct sockaddr*)&server,sizeof(server))==-1)
    // {
    //     fprintf(stdin, "cant bind name to socket.\n");
    //     return -1;
    // }
    alen = sizeof(struct sockaddr_in);

      //printf("get socket name");
      fflush(stdout);
      //getsockname(tcp_socket, (struct sockaddr*)&server, &alen);
    ip_addr_struct.s_addr = server.sin_addr.s_addr;
    sprintf(tcpAddr, "%s", inet_ntoa(ip_addr_struct));
    printf("Here: %s\n", tcpAddr);
    //printf("%s",tcpAddr);
    sprintf(tcpPort, "%d", server.sin_port);
    //printf("%d",server.sin_port);

    memset(&msg, 0, sizeof(msg));

    msg.type = REGISTER;

    char data[BUFFLEN] = {0};
    strcpy(data, PeerName);
    strcat(data, ";");
    strcat(data, Filename);
    strcat(data, ";");
    strcat(data, tcpAddr);
    strcat(data, ";");
    strcat(data, tcpPort);
    strcpy(msg.data, data);
    printf("Here is the data: %s\n", msg.data);

    write(UdpSocket, &msg, sizeof(msg));
    fflush(stdout);
    read(UdpSocket, Response, sizeof(Response));
    fflush(stdout);
    return tcp_socket;
}

/**********************************************************************
*****
* Register new content
**********************************************************************
*****/
void RegisterContent(int Socket, char* PeerName, struct sockaddr_in
reg_addr)
{
    struct PDU msg, resp;

    printf("File name to register: ");
    fflush(stdout);
    char Filename[25] = {0};

    int outLen = 0;
    outLen = read(0, Filename, sizeof(Filename));
    Filename[outLen-1] = 0;
    struct stat info;
    if(access(Filename, F_OK) != 0)
    {
        printf("Unable to find file. \n");
      fflush(stdout);
        return;
    }

    fflush(stdout);
    int tcpSocket = InitializeTCPUpload(Socket, PeerName, Filename, &resp,
reg_addr);
    if(tcpSocket < 0)
    {
        return;
    }
    if(resp.type == (char)ERR)
    {
        printf("Can't register data. \n");
          fflush(stdout);
    }
    else if(resp.type == (char)ACK)
    {
       memset(&resp, 0, sizeof(resp));
       memset(&msg, 0, sizeof(msg));

       //listen(tcpSocket, 5);
       struct sockaddr_in client;
       int clientlen = sizeof(client);
       int newTcpSocket=0;
       //newTcpSocket = accept(tcpSocket, (struct sockaddr *)&client,
&clientlen);
    //    if((newTcpSocket)<0)
    //    {
    //         fprintf(stdout, "can't accept client\n");
    //         fflush(stdout);
    //         return;
    //    }
    //    printf("created socket.\n");
    //    fflush(stdout);
    //    switch(fork()){
    //     case 0:
    //         (void) close(tcpSocket);
    //         read(newTcpSocket, &resp, sizeof(resp));
    //         if(resp.type == (char)DOWNLOAD)
    //         {
    //             FILE *file;
    //             file = fopen(Filename, "rb");
    //             msg.type = (char)DATA;
    //             while(fgets(msg.data, BUFFLEN, file)>0){
    //                 sleep(0.01);
    //                 write(newTcpSocket, &msg, sizeof(msg));
    //             }
    //             fclose(file);
    //         }
    //         (void) close(newTcpSocket);
    //         exit(0);
    //     default:
    //         (void) close(newTcpSocket);
    //         break;
    //     case -1:
    //         fprintf(stderr, "fork: error \n");
    //    }
    }
}

/************************************************************************
*****
* Initialize TCP Connection and returns tcp client socket
************************************************************************
*****/
int InitializeTCPDownload(char* Address, char* Port)
{
 // Make tcp connection
 int tcp_socket;
 struct sockaddr_in server;
 struct hostent *host;
 //char *serverHost = "0";
 printf("This is the address: %s\n", Address);
 printf("This is the port: %s\n", Port);
printf("initializing download\n");
 if ((tcp_socket = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
 fprintf(stderr, "Can't create a socket\n");
 exit(1);
 }
 bzero((char*)&server, sizeof(struct sockaddr_in));
 server.sin_family = AF_INET;
 int port = atoi(Port);

 server.sin_port = port;
 printf("%d", server.sin_port);
 if (host = gethostbyname(Address)){
 bcopy(host->h_addr, (char *)&server.sin_addr, host->h_length);
 }
 else if (inet_aton(Address, (struct in_addr *) &server.sin_addr)){
 fprintf(stderr, "Can't get server's address\n");
 }

 if (connect(tcp_socket, (struct sockaddr *)&server, sizeof(server)) ==
-1){
 fprintf(stderr, "cant connect to server: %s\n", host->h_name);
 }
printf("done init download\n");
 return tcp_socket;
}

/**************************************************************************
*****
* Download Content
***************************************************************************
*****/
void DownloadContent(int Socket, char *SelfPeerName, struct sockaddr_in
reg_addr)
{
   int n;
   printf("Enter name of content to download: \n");
   int outLen = 0;
   char Filename[50] = {0};
   outLen = read(0, Filename, sizeof(Filename));
   Filename[outLen-1] = 0;
   char peerName[50] = {0};

   printf("Enter name of peer to download from: \n");
   outLen = read(0, peerName, sizeof(peerName));
   peerName[outLen-1]=0;

   struct PDU msg;
   memset(&msg, 0, PACKETSIZE);
   msg.type = (char) SEARCH;
   strcpy(msg.data, peerName);
   strcat(msg.data, ";");
   strcat(msg.data, Filename);
```

```
    write(Socket, &msg, sizeof(msg));

    struct PDU resp;
    read(Socket, &resp, sizeof(resp));

    char temp[BUFFLEN] = {0};
    strcpy(temp, resp.data);

    //memset(peerName, 0, sizeof(peerName));
    char *contentName = strtok(temp, ";");
    char *tmpPeerName = strtok(NULL, ";");
    char *addr = strtok(NULL, ";");
    char *port = strtok(NULL, ";");

    int tcpSocket = InitializeTCPDownload(addr, port);

    memset(&msg, 0, PACKETSIZE);
    memset(&resp, 0, PACKETSIZE);

    msg.type = (char) DOWNLOAD;
    strcpy(msg.data, SelfPeerName);
    strcat(msg.data, ";");
    strcat(msg.data, contentName);
     printf("Message_Data: %s\n", msg.data);
    write(tcpSocket, &msg, sizeof(msg));

    FILE *file = 0;
    file = fopen(contentName, "wb");

    int contentExists = 1;
    while (1) {
            n = recv(tcpSocket, &resp, PACKETSIZE, 0);
            resp.data[BUFFLEN] = '\0';
            if (resp.type == ERR){
                  printf("ERROR: %s\n", resp.data);
                  remove(contentName);
                  return -1;
            }
            fprintf(file, "%s", resp.data);    // Write to file
            if (resp.type  == 'F'){            // When final pdu received,
break loop
                  break;
            }

      }
      fclose(file);
```

```
   if(contentExists)
   {
        RegisterContent(Socket, SelfPeerName, reg_addr);
   }
}

/**************************************************************************
*****
* Search for content
**************************************************************************
*****/
void SearchContent(int Socket, char *SelfPeerName)
{
    printf("Item to search for: \n");
    char content[50] = {0};
    int outLen = read(0, content, sizeof(content));
    content[outLen-1]=0;

    struct PDU msg;
    memset(&msg, 0, sizeof(msg));

    msg.type = (int) SEARCH;
    strcpy(msg.data, SelfPeerName);
    strcat(msg.data, ";");
    strcat(msg.data, content);

    write(Socket, &msg, sizeof(msg));

    struct PDU resp;
    read(Socket, &resp, sizeof(msg));

    char *tok = strtok(resp.data, ";");
    printf("Content Name: %s\n", tok);
    tok = strtok(NULL, ";");
    printf("Peer Name: %s\n", tok);
    tok = strtok(NULL, ";");
    printf("Address: %s\n", tok);
    tok = strtok(NULL, ";");
    printf("Port: %s\n", tok);
}

/**************************************************************************
*****
* Deregister Content
**************************************************************************
*****/
```

```
void DeregisterContent(int Socket, char *SelfPeerName)
{
    printf("Item to deregister: \n");
    char content[50] = {0};
    int outLen = read(0, content, sizeof(content));
    content[outLen-1]=0;

    struct PDU msg;
    memset(&msg, 0, sizeof(msg));

    msg.type = (int) DEREGISTER;
    strcpy(msg.data, SelfPeerName);
    strcat(msg.data, ";");
    strcat(msg.data, content);

    write(Socket, &msg, sizeof(msg));

    struct PDU resp;
    read(Socket, &resp, sizeof(msg));

    if(resp.type == (char) ACK)
    {
        printf("Content deregistered. \n");
    }
    else if(resp.type == (char) ERR)
    {
        printf("Unable to deregister content. \n");
    }
}

/**************************************************************************
*****
* Quit Application
**************************************************************************
*****/
int Quit(int Socket, char *SelfPeerName)
{
    struct PDU msg;
    memset(&msg, 0, sizeof(msg));
    msg.type = (char) QUIT;
    strcpy(msg.data, SelfPeerName);
    write(Socket, &msg, sizeof(msg));

    struct PDU resp;
    memset(&resp, 0, sizeof(resp));
    read(Socket, &resp, sizeof(resp));
    if(resp.type == (char)QUIT)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

/***************************************************************************
*****
* Handle Client
***************************************************************************
*****/
void handle_client(int sd)
{
      struct PDU rpdu;
      struct PDU 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);
      }
    printf("%s\n", rpdu.data);
    char temp[BUFFLEN] = {0};
    strcpy(temp, rpdu.data);

    //memset(peerName, 0, sizeof(peerName));
    char *peerName = strtok(temp, ";");
    char *contentName = strtok(NULL, ";");
      if (rpdu.type == DOWNLOAD){
            memcpy(fileName, 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 = ERR;

                  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
*************************************************************************
*****/
void sendFile(int sd, FILE *p, int fileByteSize)
{
        struct      PDU packet;
        char  fileData[BUFFLEN] = {0};
        int   n, bytesSent, totalBytesSent = 0;

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

            memcpy(packet.data, fileData, BUFFLEN);

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

/************************************************************************
*****
* main
```

```
****************************************************************************
*****/
int main ( int argc, char **argv )
{
      char  *host = "localhost";
      int   port = 3000;
      char  now[100];        /* 32-bit integer to hold time    */
      struct hostent   *phe; /* pointer to host information entry    */
      struct sockaddr_in sin, reg_addr, client;     /* an Internet
endpoint address       */
    int reg_len, client_len, ret_sel;
      int   s, n, type, new_sd, sd;      /* socket descriptor and socket
type  */

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

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

    /* Map host name to IP address, allowing for dotted decimal */
        if ( phe = gethostbyname(host) ){
                memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
        }
        else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
            fprintf(stderr, "Can't get host entry \n");
        printf("%d\n",sin.sin_addr.s_addr);
    /* Allocate a socket */
        s = socket(AF_INET, SOCK_DGRAM, 0);
        if (s < 0)
            fprintf(stderr, "Can't create socket \n");


    /* Connect the socket */
        if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
```

```
            fprintf(stderr, "Can't connect to %s \n", host);

     /* 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));

         struct ifreq ifr;
         char array[] = "enp0s3";

       reg_addr.sin_family = AF_INET;
         strncpy(ifr.ifr_name , array , IFNAMSIZ - 1);
         ioctl(sd, SIOCGIFADDR, &ifr);
       reg_addr.sin_port = htons(0);
       reg_addr.sin_addr.s_addr = inet_addr(inet_ntoa(( (struct sockaddr_in
*)&ifr.ifr_addr )->sin_addr));
         fflush(stdout);
         printf("%s\n", inet_ntoa(( (struct sockaddr_in *)&ifr.ifr_addr
)->sin_addr));
         fflush(stdout);
       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);
         printf("%d\n", reg_addr.sin_port);

     /* 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;

     char peerName[10] = {0};
     GetPeerName(peerName);

     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);

                  }
            }
            if(FD_ISSET(fileno(stdin), &rfds)) {
                  char input[10] = {0};

        printf("Enter the letter associated with the action you would like
to complete: \n");
        printf("Register content (r). \n");
        printf("List content (o). \n");
        printf("Download content (d). \n");
        printf("Search for content (s).\n");
        printf("Deregister content (t).\n");
        printf("Quit (q).\n");
        printf("\nMake a selection: ");
      fflush(stdout);
        read(0, input, 2);

        switch((int) input[0])
        {
            case REGISTER:
                RegisterContent(s, peerName, reg_addr);
                break;
            case ONLINE:
                RequestOnlineContent(s);
                break;
            case DOWNLOAD:
                DownloadContent(s, peerName, reg_addr);
                break;
            case SEARCH:
                SearchContent(s, peerName);
                break;
            case DEREGISTER:
                DeregisterContent(s, peerName);
                break;
            case 113:
                if(Quit(s, peerName))
                {
                    goto Exit;
                }
        }

            }

    }
Exit:
    close(s);
    exit(0);
}
```