Table of Content

1. Introduction

For the COE768 final project, a P2P application was built which allowed users to share content with each other through an index server. Every client could both provide and request content. Content servers registered their data with the index server, while the content clients received the address of that data so they could access it from the content server. UDP was used for communication between peers and the index server and TCP was used to download content between peers. The protocol data unit (PDU) format that was used consisted of 8 PDU types; content registration, content download, content search, de-registration, content data, on-line registered content, acknowledgement and error.

Socket programming is a fundamental part of the project as it allows for communication between peers and the index server through a socket. In communication from peer to peer, the content client establishes a socket to receive the data while the content server establishes a socket to provide the content from. The content server binds to the client socket using its IP and port number and listens in passive mode until a connection request is obtained. This will enable data to begin transferring. When addressing the socket programming in regards to the index server and the peers, they do not require socket acknowledgement or connection request, rather two forms of function calls were used; receive from and send to. This provided addressing for communication.

2. Description of the client and server programs

In the application, the client and server are the primary members for the socket connection. There were 5 main objectives that needed to be completed and demonstrated in the application; content registration, content download, content online listing, content de-registration and quit.

2.1 Client

The client was tasked with ensuring that the 5 PDUs are displayed and functioning correctly in the application. The client code allows a file to be uploaded to the server and then downloaded back through the socket connection. This can only be done by connecting to the server directory socket (sd) which is the socket connection between the server and client. The program verifies that the connection has been successful by receiving an acknowledgement from the server using echo functions.

2.2 Server

The server code operates similarly to the client however the purpose for the server is to establish a LAN connection between any two users in order to exchange information. The PDUs in the server code and client code will be the same however, when a response is received from the client, it must be sent to the server side to confirm that all actions were successful or an error has occurred.

2.3 Protocol Data Unit Types

2.3.1 Content Registration [R]

For the content registration PDU, we are working with the UDP protocol between peers and the index server. We use the PDU type R and send data consisting of the peer name, file name and the addressing of the content. If there exists a peer of the same name with the filename, the index server is expected to send back a PDU E type packet, prompting the peer of an error with the peer name. Otherwise, if the peer name is good the index server responds with an acknowledgment packet. In terms of content registration peers are allowed to register numerous files to the index server. Files are specified to have their own unique sockets so in the later portions of the process, when data is downloaded it is simple to establish TCP connections to the desired socket for the files.

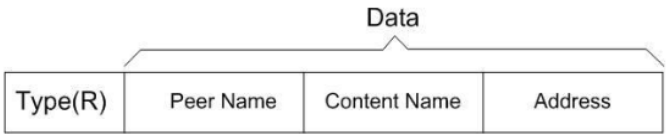


Figure 1: R Type Data Packet

```
256     tpdu.PDU_type = 'R';                                                    // Set PDU Type 'R' for registration
257     memcpy(&tpdu.PDU_data[FILENAMESIZE], name, FILENAMESIZE);               // Copy content name to PDU data
258     memcpy(&tpdu.PDU_data[2*FILENAMESIZE],&reg_addr.sin_port,sizeof(reg_addr.sin_port));    // Copy port to PDU data
259
260     while(1){
261         memcpy(tpdu.PDU_data, user, FILENAMESIZE);                          // Copy user to PDU data
262
263         // Error Message
264         if( (i=write(s_sock, &tpdu, sizeof(tpdu))) <=0 ){
265             printf("[!] ERROR: Registration Write Error\n");
266             return;
267         }
268
269         // Error Message
270         if ((i=read(s_sock, &recvPDU, sizeof(recvPDU))) <0){
271             printf("[!] ERROR: Registration Read Error\n");
272             return;
273         }
274
275         if(recvPDU.PDU_type == 'A'){                // Check if the recieved PDU type is an Acknoledgement
276             FD_SET(p_sock, &afds);                  // Listening on TCP socket
277             table[p_sock].status = 1;               // Set status for registration to 1
278             strcpy(table[p_sock].name,name);        // Copy filename to table
279
280             if(nfds <= p_sock){
281                 nfds = p_sock+1;                    // Updates the max file descripton when necessary
282             }
283
284             printf("[+] Registration Name: %s\n",name);
285             printf("[+] Registration Port: %d\n\n", ntohs(reg_addr.sin_port));
286             return;
287         }
288
289         if(recvPDU.PDU_type == 'E'){
290             printf("[!] ERROR: Please choose a different Filename\n");
291             scanf("%s",user);
292         }
293     }
```

Figure 2: Registration code snippet from client.c

In the above snippet of code we structure our temporary PDU packet to reflect the form of Figure 1 using lines 256 to 261. We start by setting the temporary PDU type to R and copying the filename, address and peer names to the data array. After structuring our data we use line 264 to send it to the index server. In the event of an error during transmission the user is prompted with an error message. After sending the contents to the server, the server sends back a packet which is stored in recvPDU. Again if there is an error with reading from the server the user is prompted with an error message. The final step in the registration portion is to check if the data type sent from the server is A for acknowledgment or E for error. If the data type is A that means the communication was successful and the user gets feedback as to the filename and port address that is being registered. In the event that there is an error the user is made aware of it.

2.3.2 Content Download [D]

The content download PDU required a few components in order for it to perform as required for the project. The first step in the process is a content client communicating with the index server in regards to searching for the content the client needs. This request is done by sending an S type packet with the client name and file name. The server responds back with either an ‘S' type packet, standing for a successful search or an ‘E’ type meaning no content match is found. Upon a successful ‘S’ type packet from the server the client extracts the content server address from the packet and sends a ‘D’ type packet to the content server to establish a TCP connection for the download. Content server sees the ‘D’ type packet which contains the filename and user name and sends back 100 byte size ‘C’ packets with the file contents, until the file is completely sent over to the content client. The last step in the content download process is that the content client now uses UDP communication to register the downloaded content with the index server and if done successfully, the server sends back an acknowledgement.

```
297  /* SEARCH CONTENT METHOD */
298  int search_content(int s_sock, char *name, PDU *recvPDU){
299
300      // Initialize variables
301      int n;
302      PDU tpdu;
303
304      tpdu.PDU_type = 'S';                              // Set PDU type to 'S'
305      memcpy(tpdu.PDU_data, name, FILENAMESIZE);        // Copy content name to PDU data
306
307      // Error Messages
308      if((n=write(s_sock, &tpdu, BUFLEN+1)) < 0){       // Indicate if there is error writing to server
309          printf("[!] ERROR: Search Content Write Error\n");
310          return -1;
311      }
312
313      if((n = read(s_sock, recvPDU, BUFLEN+1))<0){      // Indicate if there is error reading from server
314          printf("[!] ERROR: Search Content Read Error\n");
315          return -1;
316      }
317
318      if(recvPDU->PDU_type == 'E'){                     // Check If the recvPDU type is E Indicating an Error
319          printf("[!] ERROR: %s - Filename can not be Found \n\n",name);
320          return -1;
321      }
322
323      if(recvPDU->PDU_type == 'S'){                     // Check If the type is S indicating Search success
324          printf("\n[+] Search results successfuly found\n");
325          return 0;
326      }
327      else{
328          printf("[!] ERROR: Search Content Protocol Error\n");
329          return -1;
330      }
```

Figure 3: Search content method from client.c

The above code is used to communicate between the content client and the index server to figure out if the content that is needed is available. On line 304 and 305 the data type is set as S and the contents is passed the filename that is to be searched. The server writes the packet on line 308, with error detection on whether the packet is sent successfully. After the content client receives back and stores the contents from the server in recvPDU. On line 318 we see that if the data type is ‘E’ this means the filename was not found. On line 323 we see that if the data type is ‘S’ this means that the file name is in fact registered in the index server.

```
353     if (connect(p_sock, (struct sockaddr *)&p_addr, sizeof(p_addr)) == -1){
354         printf("[!] ERROR: Download can't connect to the remote peer\n");
355         return -1;
356     }
357 
358     tpdu.PDU_type = 'D';                                        // Set PDU type to 'D'
359     write(p_sock, &tpdu, 1);                                    // Writing 1 bite from the pointer to tpdu to p_sock
360     fd = open(name, O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);       // Open the file for reading and writing
361 
362     if(fd < 0){                                 // Check if the file opens successfuly
363         printf("[!] ERROR: Download can't open file\n");
364         return -1;
365     }
366 
367     n=read(p_sock, &recvPDU, sizeof(PDU));      // Reading from recvPDU to socket
368 
369     if(recvPDU.PDU_type == 'C'){                // Check if the type is 'C'
370         write(fd, recvPDU.PDU_data, n-1);       // Write data from the recvPDU to file
371 
372         while((n=read(p_sock, recvPDU.PDU_data, BUFLEN))>0){  // While loop handles files larger than 100 Bytes
373             write(fd, recvPDU.PDU_data, n);
374         }
375     }
376     else{
377         if(recvPDU.PDU_type == 'E')             // Check If the recvPDU type is E Indicating an Error
378             printf("[!] ERROR: Download content iin unavailable for remote peer\n");
379         else
380             printf("[!] ERROR: Download Protocol Error\n");
381             close(fd);                          // Close File
382             close(p_sock);                      // Close Peer Socket
383             return -1;
384     }
385 
386     close(fd);                                  // Close File
387     close(p_sock);                              // Close Peer Socket
388     return 0;
```

Figure 4: Continuation of search content method from client.c

After knowing the content is registered we connect to the content server using line 353 function call connect and the address we obtain from the index server. Then we write to the peer server socket the temporary PDU packet with the ‘D’ type message. After this we also have to open a file to store the downloaded content, so we create a file if it is not already present in the content client that is both read and write for the data to be written into. Lines 367 to 375 the content client reads the size of the ‘C’ type packets sent from the content server and writes this into the file. If at any point there is an issue and a ‘E’ type is received or there is an issue with the protocol the content client is prompted with an error message. After the contents are completely downloaded both the file and content server socket are closed on lines 386 and 387.

```
182     /* PDU Type 'D' - Download Content  */
183     if(typePDU=='D'){
184         printf("\n[>] Enter the File Name you wish to Download:\n");
185         scanf("%s",name);                                   //Filename input
186         if(search_content(s_sock,name, &recvPDU) == 0){     //Checking if filename exists
187             if(client_download(name, &recvPDU) == 0){       //Complete Download to Client
188                 registration(s_sock,name);                  //Automatically register Client
189             }
190         }
191         continue;
192     }
```

Figure 5: Download content method from client.c

This portion of code shows us that after successfully completing the search for the contents, and downloading of contents. We then run the registration method to make the content client automatically become a content server for the downloaded contents.

2.3.3 Online-Listing of Content [O]

This Function is used by a peer to communicate with the index server to find out the contents that are currently registered. A ‘O’ type PDU is sent from the peer to the index server and the index server responds with a ‘O’ type PDU with the list of registered files in the data field.

```
391  /* LISTING METHOD */
392  void listing(int s_sock){
393
394      // Initialize Variables
395      PDU tpdu;
396      tpdu.PDU_type = 'O';                        // Set PDU tpye to 'O'
397
398      write(s_sock, &tpdu, sizeof(tpdu));         // writing from the pointer tpdu to the server socket
399      read(s_sock, &recvPDU, sizeof(tpdu));       // Read response received PDU from the server socket
400
401      printf("\n[+] On-line Content List:\n%s\n\n", recvPDU.PDU_data);
402      return;
403  }
```

Figure 6: Online Content Listing method from client.c

We can see that the above code sets the temporary PDU type to O on line 396. This is then written to the index server using UDP protocol on line 398. After the server receives the ‘O’ PDU it sends back a ‘O’ PDU which is stored into the recvPDU on line 399. Lastly we can see on line 401 we print the string of data from the recvPDU to the terminal for the user to see.

2.3.4 Content Deregistration [T]

The de-registration data type allows a content server to unregister a file that has been previously stored in the registered files table. The content server sends a ‘T’ type PDU to the index server and upon successful removal the server sends back a ‘A’ type PDU acknowledging that the function worked successfully.

```
440     tpdu.PDU_type = 'T';                        // Set the PDU type to 'T'
441     strcpy(tpdu.PDU_data, name);                // Copy the file name to PDU data
442     strcpy(&tpdu.PDU_data[FILENAMESIZE], user); // Copy user name to PDU data
443     write(s_sock, &tpdu, sizeof(tpdu));         // Write from the tpdu pointer to the server socket
444     read(s_sock, &recvPDU, sizeof(recvPDU));    // Read Servers Response
445
446     // Check is De-Registration was successful
447     if(recvPDU.PDU_type == 'T'){
448         printf("[+] De-Registration Content %s is successfully de-registered\n", name);
449         return;
450     }
451     else{
452         printf("[!] ERROR: De-Registration content %s is not registered\n",name);
453         return;
454     }
455 }
```

Figure 7: Registration code snippet from client.c

Prior to this code we have error detection for checking whether the file is not registered, this will prompt an error. We set the temporary PDU to type ‘T’ and copy the name of the file we want to remove to the data element on line 441. After we copy the user name to the PDU on 442. This information is written to the index server using the write function and a received PDU is returned through the read function and stored to the recvPDU on line 444. When we czech the recvPDU data type if it is of type ‘T’ this means that the index server has successfully removed the file and returned a ‘T’ type packet. If we did not receive a ‘T’ type PDU we prompt the user with an error message on line 452.

2.3.5 Quit [Q]

In the event that a user wants to terminate its connection with the index server there are a few steps that need to take place. The user must de-register all of its contents on the index server and then it is allowed to close the terminal. This is done by sending ‘T’ type PDUs to the index server removing the files it has registered.

```
208         /* PDU Type 'Q' - Quit */
209         if(typePDU == 'Q'){
210             quit(s_sock);
211             exit(1);
212         }

457 /* QUIT METHOD */
458 void quit(int s_sock)
459 {
460     int i;
461     for(i=3; i<nfds; i++){
462         if(table[i].status == 1)
463             deregistration(s_sock,table[i].name);
464         }
465     return;
466 }
```

Figure 8: Quit method from client.c

The following code is used for deregistration, and once registration is complete we execute the exit function call to terminate the terminal.

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

Table 1: Summary of the function and direction of the PDU types used in this project

3. Observations and analysis

Figure 9: Running the three peers and the index server

Figure 10: Peer 1 enters the 'i' command to display the menu of possible commands to enter into the terminal.

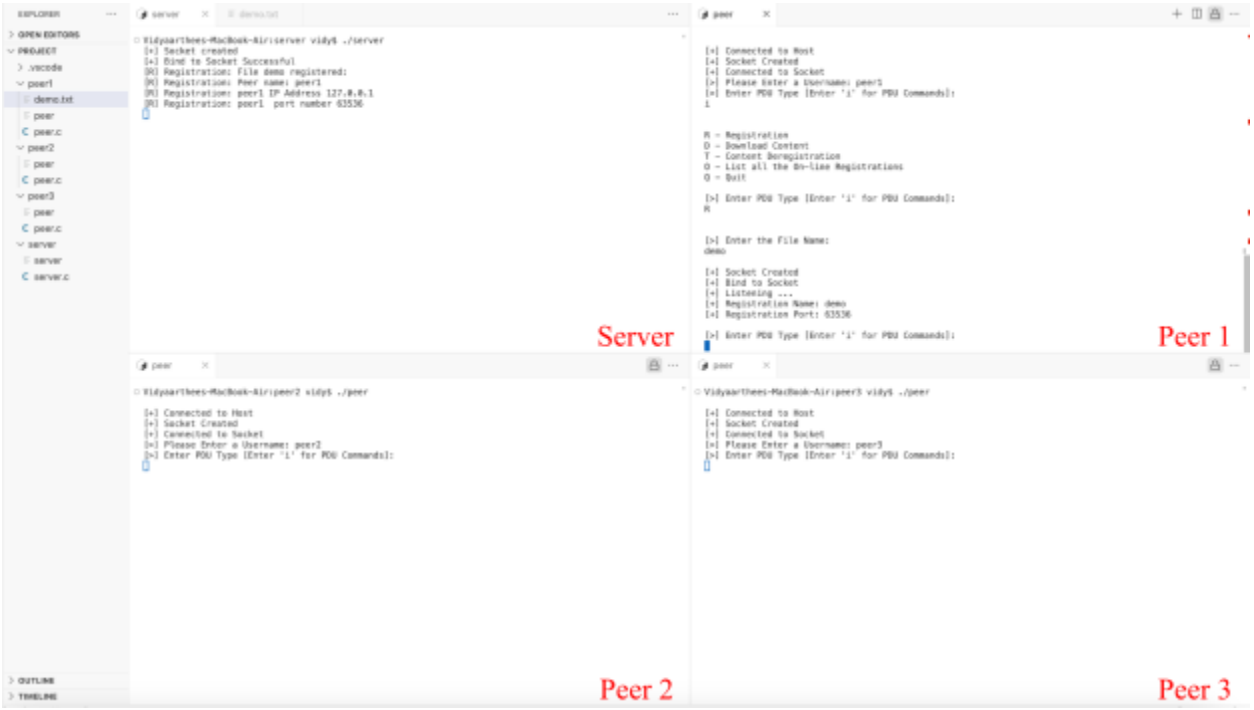


Figure 11: Peer 1 registers the demo file to the index server both the server and peer 1 receive confirmation with the registered file name and the port address.

Figure 12: Peer 2 requests a list of the contents registered to the index server. The server displays a prompt stating a request was made. The peer receives a list of the files that have been registered. In this example we can see the demo.txt file is registered.

Figure 13: Peer2 downloads the demo file from peer1. We see that peer2 is prompted that the file is successfully located and then the download is completed.

Figure 14: Peer 3 does a search of the registered files and sees the demo file

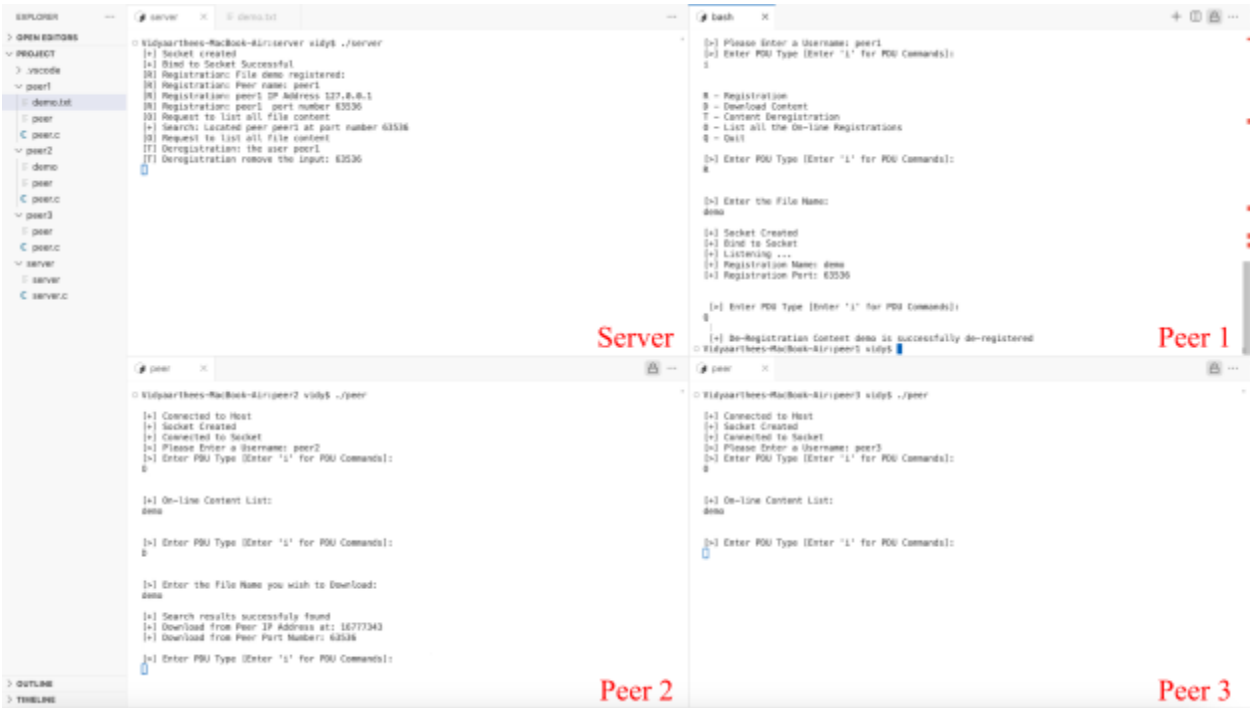


Figure 15: Peer 1 quits and we see that the contents registered by peer1 are deregistered. Peer 1 receives a prompt that demo.txt was successfully de-registered.

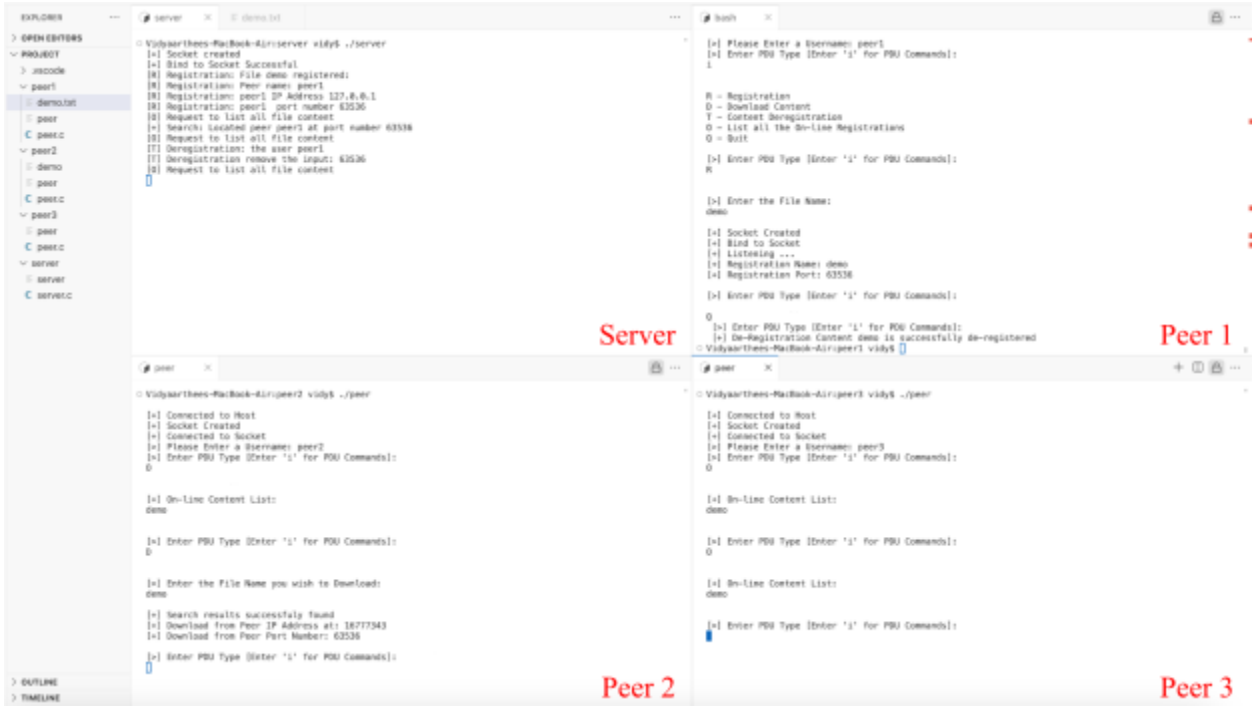


Figure 16: Peer 3 does another search of the contents on the index server. We see that the demo file is still registered. This is because peer 2 automatically registered the demo file after it completed the download earlier in this experiment.

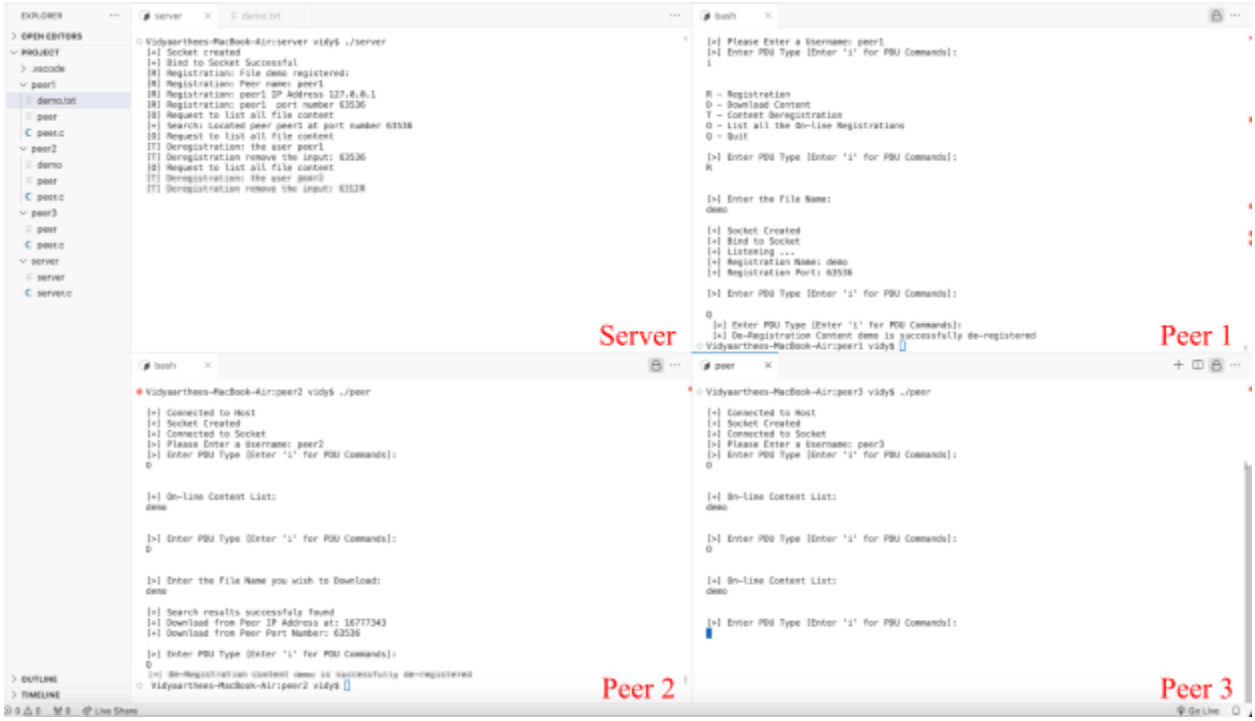


Figure 17: Peer 2 terminates its session and we see that the server de-registers the contents that peer2 was being the content server for. On the user side peer2 is prompted with a successful de-registration

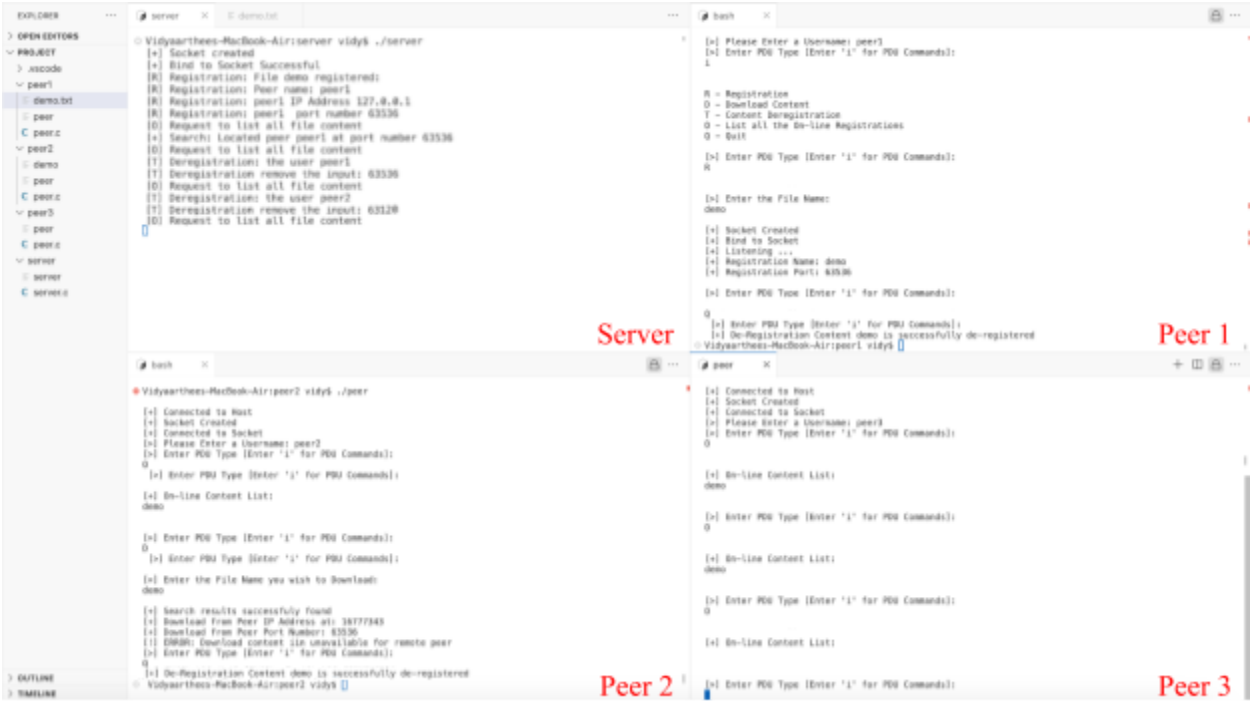


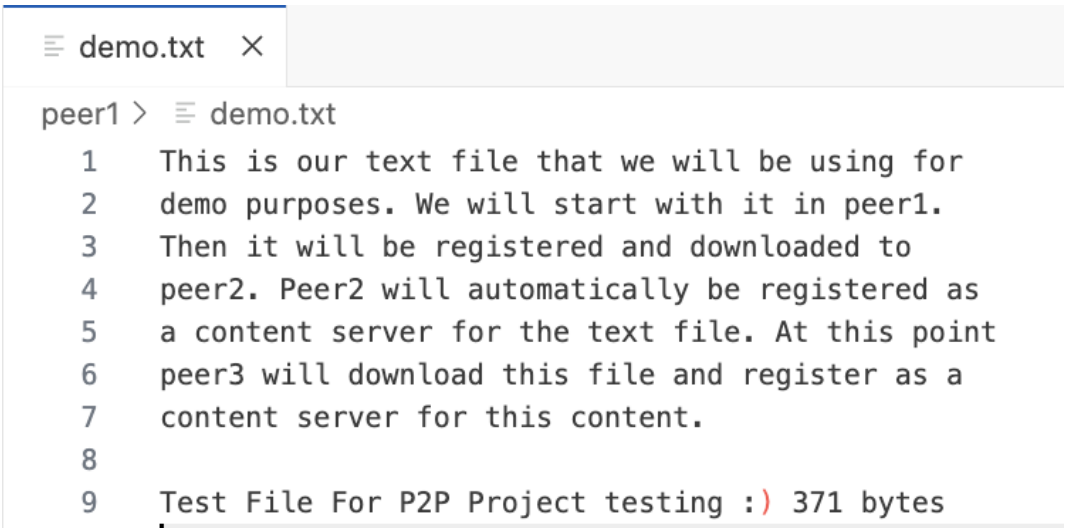
Figure 18: For the final figure we see that when peer 3 lists the registered files again, now there are no files available. So if we attempt to download there will be an error.

4. Conclusions

The project was successful in performing all of the PDU types needed to make the peer to peer project function. This project allowed us to combine our knowledge learned in earlier labs on both TCP and UDP protocols. In the previous labs, we worked with servers and clients to practice the functionality of UDP and TCP. This knowledge provided us with the fundamentals to program the index servers, content servers and content clients. We communicate between the peers and servers using UDP and we perform data transfer by establishing TCP connections for communication. We use our knowledge of structures to structure the PDU types with the particular contents and data needed to make each data type function effectively. We limit the maximum writing size of a file to 100 bytes. By taking this into account we send multiple content packets for larger files. There are many portions of this project and in order to make sure everything works properly we constantly used acknowledgement messages or error prompts to troubleshoot any possible issues with the performance of the project. We learned a lot from the completion of this major project which can serve as a stepping stone for future projects. In the future we would like to add the ip address into the addressing portions so that we can communicate between multiple workstations or networks since we used LAN to establish communications through sockets on one computer throughout this lab.

5. Appendix

5.1 Text File Contents



5.2 Peer.c Code

```c
/*  COE 768 - Final Project
    -----
    Name(s): Maria Labeeba (500960386)
             & Vidy Matadeen (501054327)
    Due Date: Nov 27th, 2023
    File: peer.c

*/

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

/* IMPORTS */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/select.h>
#include <errno.h>

/* DEFINITIONS */
#define QUIT "quit"                 // Define string 'quit' for quitting program
#define SERVER_PORT 15000           // Define Port of server
#define BUFLEN  100                 // Define length of PDU_data
#define FILENAMESIZE 10             // Define max size for file name
#define CONNECTIONSIZE  100         // Define max number of connections

/* STRUCT: PDU */
typedef struct {
    char    PDU_type;               // Character to represent PDU type
    char    PDU_data[BUFLEN];       // Array stores data in PDU
} PDU;
PDU recvPDU;                        // Recieved PDU

struct {
    int     status;                 // Integer to represent the status bit of file
    char    name[FILENAMESIZE];     // Array stores filename connected
} table[CONNECTIONSIZE];            // Keeps track of registered content

char user[FILENAMESIZE];            // Array stores user name with max size of FILENAMESIZE

/* DESCRIPTORS */
int s_sock;                         // Socket Descriptor
int fd;                             // File  Decriptor
int nfds;                           // Max Number of Files Descriptor
fd_set rfds, afds;                  // Setting Reading and Active File Descriptors

/* FUNCTIONS */
void    registration(int, char *);              // Content Registration
int     search_content(int, char *, PDU *);     // Content Search
int     client_download(char *, PDU *);         // Content Download by Client
void    server_download();                      // Content Download by Server
void    deregistration(int, char *);            // Content Deregistration
void    listing(int);                           // Listing Online Content
void    quit(int);                              // Quiting Program

/* Main Functions */
int main(int argc, char **argv){
    int s_port = SERVER_PORT;                   // Default server port 15000
    int n;                                      // Used in For - Loops
    int alen = sizeof(struct sockaddr_in);      // Length of Socket Address
    struct  hostent *hp;                        // Host input pointer
    struct  sockaddr_in server;                 // ServSocket address structure
    char typePDU, name[FILENAMESIZE];           // For user input
    char *host = "localhost";                   // Default host name
    struct sigaction sa;                        // Signal Action Structure


```

```
/* Switch Case for Command Line Handling*/
switch(argc){
    case 1:
        break;
    case 2:
        host = argv[1];
        break;
    case 3:
        host = argv[1];
        s_port = atoi(argv[2]);
        break;
    default:
        printf(stderr, "[-] Usage: %s host [port]\n", argv[0]);
        exit(1);
}

/* UDP Connection with the index server */
memset(&server, 0, alen);                   // Server address sockets set to 0
server.sin_family = AF_INET;
server.sin_port = htons(s_port);

/* Establishing Host Connection */
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("\n[!] Can NOT connect to host \n");
        exit(1);
}
printf("\n[+] Connected to Host\n");

/* Establishing Socket Connection */
s_sock = socket(PF_INET, SOCK_DGRAM, 0);
if (s_sock < 0){
        printf("[!] Can NOT create Socket\n");
        exit(1);
}
printf("[+] Socket Created\n");

if (connect(s_sock, (struct sockaddr *)&server, sizeof(server)) < 0){
    printf("[!] Can NOT connect to Socket\n");
    exit(1);
}
printf("[+] Connected to Socket\n");
```

```
/* User Name Configuration */
printf("[>] Please Enter a Username: ");
scanf("%s",user);

/* Initialization of SELECT structure and table structure */
FD_ZERO(&afds);
FD_SET(s_sock, &afds);                      // Listening on TCP socket
FD_SET(0, &afds);                           // Listening on stdin
nfds = 1;

for(n=0; n<CONNECTIONSIZE; n++)             // Set all status values to -1
    table[n].status = -1;


/* Main Loop */
while(1){

    printf("[>] Enter PDU Type [Enter 'i' for PDU Commands]:\n ");
    memcpy(&rfds, &afds, sizeof(rfds));


    // If there is not input, return Error
    if (select(nfds, &rfds, NULL, NULL, NULL) == -1){
        printf("[!] Error: %s\n",strerror(errno));
        exit(1);
    }

    // If there is an input, read character as PDU Type from user
    if (FD_ISSET(0, &rfds)) {
        typePDU = getchar();

        /* PDU Type Command Options */
        if(typePDU == 'i'){
            printf("\nR - Registration\n");
            printf("D - Download Content\n");
            printf("T - Content Deregistration\n");
            printf("O - List all the On-line Registrations\n");
            printf("Q - Quit\n\n");
            continue;
        }
```

```c
            /* PDU Type R - Registration */
            if(typePDU == 'R') {
                printf("\n[>] Enter the File Name:\n");
                scanf("%s",name);                                   //Filename input
                registration(s_sock, name);                         //name passed to registration function
                continue;
            }

            /* PDU Type 'D' - Download Content  */
            if(typePDU=='D'){
                printf("\n[>] Enter the File Name you wish to Download:\n");
                scanf("%s",name);                                   //Filename input
                if(search_content(s_sock,name, &recvPDU) == 0){     //Checking if filename exists
                    if(client_download(name, &recvPDU) == 0){       //Complete Download to Client
                        registration(s_sock,name);                  //Automatically register Client
                    }
                }
                continue;
            }

            /* PDU Type O - List on-line Content */
            if(typePDU == 'O'){
                listing(s_sock);
                continue;
            }

            /* PDU Type 'T' - Content Deregistration */
            if(typePDU == 'T'){
                printf("\n[>] Enter the Filename you wish to Deregister:\n");
                scanf("%s",name);
                deregistration(s_sock,name);
                continue;
            }

            /* PDU Type 'Q' - Quit */
            if(typePDU == 'Q'){
                quit(s_sock);
                exit(1);
            }
        }
        /* Server to client File Transfer*/
        server_download(s_sock);
    }
return 0;
}


/* REGISTRATION METHOD */
void registration(int s_sock,char *name){

    // Initialize variables
    struct sockaddr_in reg_addr, addr;      // Socket address structure
    int p_sock, alen, i;                    // Peer socket discriptor, socket length address struct
    int p_port = 0;                         // peer port
    PDU tpdu, recvPDU;                      // PDU structure for sending and recieving data

    alen = sizeof(struct sockaddr_in);      // Gets the size of the socket


    // Create a Peer socket
    if ((p_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        printf("[!] ERROR: Registration Can NOT create socket\n");
        return;
    }
    printf("\n[+] Socket Created\n");

    // Bind to Peer Socket
    memset(&reg_addr, 0, sizeof(struct sockaddr_in));   // Initialize sockaddr_in
    reg_addr.sin_family = AF_INET;                      // Set address to AF_INET
    reg_addr.sin_port = htons(0);                       // Set the port to 0
    reg_addr.sin_addr.s_addr = htonl(INADDR_ANY);       // Allows socket to bind to available local IP address

    if (bind(p_sock, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) == -1){
            printf("[!] ERROR: Registration Can NOT bind to socket\n");
            return;
    }
    printf("[+] Bind to Socket\n");                     // Content Server Bound

    listen(p_sock,5);
    printf("[+] Listening ...\n");

    getsockname(p_sock, (struct sockaddr *) &reg_addr, &alen);                          //retrieves the current name for the specified p socket descriptor

    tpdu.PDU_type = 'R';                                                                // Set PDU Type 'R' for registration
    memcpy(&tpdu.PDU_data[FILENAMESIZE], name, FILENAMESIZE);                           // Copy content name to PDU data
    memcpy(&tpdu.PDU_data[2*FILENAMESIZE],&reg_addr.sin_port,sizeof(reg_addr.sin_port));    // Copy port to PDU data

    while(1){
        memcpy(tpdu.PDU_data, user, FILENAMESIZE);                                      // Copy user to PDU data

        // Error Message
        if( (i=write(s_sock, &tpdu, sizeof(tpdu))) <=0 ){
            printf("[!] ERROR: Registration Write Error\n");
            return;
        }
```

```
268
269         // Error Message
270         if ((i=read(s_sock, &recvPDU, sizeof(recvPDU))) <0){
271             printf("[!] ERROR: Registration Read Error\n");
272             return;
273         }
274
275         if(recvPDU.PDU_type == 'A'){                // Check if the recieved PDU type is an Acknoledgement
276             FD_SET(p_sock, &afds);                  // Listening on TCP socket
277             table[p_sock].status = 1;               // Set status for registration to 1
278             strcpy(table[p_sock].name,name);        // Copy filename to table
279
280             if(nfds <= p_sock){
281                 nfds = p_sock+1;                    // Updates the max file descripton when necessary
282             }
283
284             printf("[+] Registration Name: %s\n",name);
285             printf("[+] Registration Port: %d\n\n", ntohs(reg_addr.sin_port));
286             return;
287         }
288
289         if(recvPDU.PDU_type == 'E'){
290             printf("[!] ERROR: Please choose a different Filename\n");
291             scanf("%s",user);
292         }
293     }
294     printf("[!] ERROR: Registration protocol error\n\n");
295 }
296
297 /* SEARCH CONTENT METHOD */
298 int search_content(int s_sock, char *name, PDU *recvPDU){
299
300     // Initialize variables
301     int n;
302     PDU tpdu;
303
304     tpdu.PDU_type = 'S';                            // Set PDU type to 'S'
305     memcpy(tpdu.PDU_data, name, FILENAMESIZE);      // Copy content name to PDU data
306
307     // Error Messages
308     if((n=write(s_sock, &tpdu, BUFLEN+1)) < 0){         // Indicate if there is error writing to server
309         printf("[!] ERROR: Search Content Write Error\n");
310         return -1;
311     }
312
313     if((n = read(s_sock, recvPDU, BUFLEN+1))<0){        // Indicate if there is error reading from server
314         printf("[!] ERROR: Search Content Read Error\n");
315         return -1;
316     }
317
318     if(recvPDU->PDU_type == 'E'){                       // Check If the recvPDU type is E Indicating an Error
319         printf("[!] ERROR: %s - Filename can not be Found \n\n",name);
320         return -1;
321     }
322
323     if(recvPDU->PDU_type == 'S'){                       // Check If the type is S indicating Search success
324         printf("\n[+] Search results successfuly found\n");
325         return 0;
326     }
327     else{
328         printf("[!] ERROR: Search Content Protocol Error\n");
329         return -1;
330     }
331 }
332
333
334 /* CLIENT DOWNLOAD METHOD */
335 int client_download(char *name, PDU *pdu){
336
337     // Initialize Variables
338     struct  sockaddr_in p_addr;
339     int p_sock, n, fd;
340     PDU tpdu, recvPDU;
341
342     memcpy(&recvPDU, pdu, sizeof(PDU));                                     // Copy content name to PDU data
343     memcpy(&p_addr, recvPDU.PDU_data, sizeof(p_addr));
344     printf("[+] Download from Peer IP Address at: %d\n", p_addr.sin_addr.s_addr);
345     printf("[+] Download from Peer Port Number: %d\n", ntohs(p_addr.sin_port));
346
347     // Error Messages
348     if ((p_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
349         printf("[!] ERROR: Download can't creat a socket\n");
350         return -1;
351     }
352
353     if (connect(p_sock, (struct sockaddr *)&p_addr, sizeof(p_addr)) == -1){
354         printf("[!] ERROR: Download can't connect to the remote peer\n");
355         return -1;
356     }
357
```

```
    tpdu.PDU_type = 'D';                                        // Set PDU type to 'D'
    write(p_sock, &tpdu, 1);                                    // Writing 1 bite from the pointer to tpdu to p_sock
    fd = open(name, O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);       // Open the file for reading and writing

    if(fd < 0){                                 // Check if the file opens successfuly
        printf("[!] ERROR: Download can't open file\n");
        return -1;
    }

    n=read(p_sock, &recvPDU, sizeof(PDU));      // Reading from recvPDU to socket

    if(recvPDU.PDU_type == 'C'){                // Check if the type is 'C'
        write(fd, recvPDU.PDU_data, n-1);       // Write data from the recvPDU to file

        while((n=read(p_sock, recvPDU.PDU_data, BUFLEN))>0){  // While loop handles files larger than 100 Bytes
            write(fd, recvPDU.PDU_data, n);
        }
    }
    else{
        if(recvPDU.PDU_type == 'E')             // Check If the recvPDU type is E Indicating an Error
            printf("[!] ERROR: Download content iin unavailable for remote peer\n");
        else
            printf("[!] ERROR: Download Protocol Error\n");
            close(fd);                          // Close File
            close(p_sock);                      // Close Peer Socket
            return -1;
    }

    close(fd);                                  // Close File
    close(p_sock);                              // Close Peer Socket
    return 0;
}

/* LISTING METHOD */
void listing(int s_sock){

    // Initialize Variables
    PDU tpdu;
    tpdu.PDU_type = 'O';                        // Set PDU tpye to 'O'

    write(s_sock, &tpdu, sizeof(tpdu));         // writing from the pointer tpdu to the server socket
    read(s_sock, &recvPDU, sizeof(tpdu));       // Read response received PDU from the server socket

    printf("\n[+] On-line Content List:\n%s\n\n", recvPDU.PDU_data);
    return;
}


/* DEREGISTRATION METHOD */
void deregistration(int s_sock,char *name){

    // Initialize Variables
    int i;
    int located = -1;
    PDU tpdu, recvPDU;

    // Checks and updates to see if the file is registered
    for(i=3; i<nfds; i++){
        if(strcmp(table[i].name, name)==0){
            if(table[i].status == 1){
                located = i;
            }
        }
    }

    // If content was not found or registered
    if(located == -1){
        printf("[!] ERROR: De-Registration Content %s is not registered\n", name);
        return;
    }
    else{
        // If content was already registered
        if(table[located].status == -1){
            printf("[!] ERROR: De-Registration Content %s is not registered\n",name);
            return;
        }

        FD_CLR(located, &afds);                 // Clear file disruptors
        if(nfds == located+1) nfds = located;   // Update Max file disruptors when necessary
            table[located].status = -1;         // If located we reset the status bit
            close(located);
    }

    tpdu.PDU_type = 'T';                        // Set the PDU type to 'T'
    strcpy(tpdu.PDU_data, name);                // Copy the file name to PDU data
    strcpy(&tpdu.PDU_data[FILENAMESIZE], user); // Copy user name to PDU data
    write(s_sock, &tpdu, sizeof(tpdu));         // Write from the tpdu pointer to the server socket
    read(s_sock, &recvPDU, sizeof(recvPDU));    // Read Servers Response

    // Check is De-Registration was successful
    if(recvPDU.PDU_type == 'T'){
        printf("[+] De-Registration Content %s is successfully de-registered\n", name);
        return;
    }
```

```
    else{
        printf("[!] ERROR: De-Registration content %s is not registered\n",name);
        return;
    }
}

/* QUIT METHOD */
void quit(int s_sock)
{
    int i;
    for(i=3; i<nfds; i++){
        if(table[i].status == 1)
            deregistration(s_sock,table[i].name);
        }
    return;
}

/* SERVER DOWNLOAD METHOD*/
void server_download(){

    // Initialize Variables
    int n, i, new_sd, alen;
    struct sockaddr_in  client;
    PDU tpdu;

    for(i=3; i<nfds; ++i){
        if (FD_ISSET(i, &rfds)) {
            new_sd = accept(i, (struct sockaddr *)&client, &alen);
            n=read(new_sd, &recvPDU, sizeof(recvPDU));

            if(recvPDU.PDU_type=='D'){                      // Check if the PDU type is 'D'
                fd = open(table[i].name,O_RDONLY);          // open file in read only mode using the file table

                if(fd >= 0){                                // Check if the file opens successfuly
                    tpdu.PDU_type = 'C';                    // Set the PDU type to 'C'
                    n = read(fd, tpdu.PDU_data, BUFLEN);    // read the data from tpdu data to the file in 100 byte size
                    write(new_sd, &tpdu, n+1);              // write to the new socket from tpdu pointer
                    while((n = read(fd, tpdu.PDU_data, BUFLEN))>0){ // While loop handles files larger than 100 Bytes
                        write(new_sd, tpdu.PDU_data, n);
                    }
                    printf("[+] Transmission Completed \n");
                }
                else{
                    tpdu.PDU_type = 'E';                    // Check if file did not open and set the PDU type to E
                    write(new_sd, &tpdu, sizeof(tpdu));     // Write to new socket the pointer from tpdu
                }

                close(fd);                                  // Close the read only file
                close(new_sd);                              // Close the socket connection
            }
        }
    }
    return;
}
```

5.3 Server.c Code

```c
/*  COE 768 - Final Project
    ----
    Name(s): Maria Labeeba (500960386)
             & Vidy Matadeen (501054327)
    Due Date: Nov 27th, 2023
    File: server.c

*/

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

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

/* DEFINITIONS */
#define BUFLEN  100             // Defines length of message buffers
#define FILENAMESIZE    10      // Defines Max size of file names
#define CONNECTIONSIZE  100     // Defines mac number of connections

/* STRUCTURE - PEER/CONTENT INPUT INFO */
typedef struct input{
    char user[FILENAMESIZE];    // Array to store usernames
    struct sockaddr_in addr;    // Structure to store the socket address info
    short token;                // Placeholder for input
    struct input *next;         // Pointer to the next input
} INPUT;


/* STRUCTURE - FILE LIST */
typedef struct{
    char name[FILENAMESIZE];    // Array to store name of file_list
    INPUT*start;                    // Pointer to the files
} FILELIST;

FILELIST file_list[CONNECTIONSIZE]; // Array with size of "CONNECTIONSIZE"
int indexMax=0;

/* STRUCTURE - REPRESENTS PDU */
typedef struct{
    char PDU_type;          // Type of PDU
    char PDU_data[BUFLEN];  // Array to store PDU data
} PDU;

PDU tpdu;                   // initialize tpdu with PDU structure

/* FUNCTION PROTOTYPES FOR SEARCH, REGISTRATION, DEREGISTRATION */

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


/* MAIN METHOD */
int main(int argc, char *argv[]){

    // Initialize variables
    struct sockaddr_in sin, *p_addr;
    char *server = "15000";
    char name[FILENAMESIZE], user[FILENAMESIZE];
    int alen = sizeof(struct sockaddr_in);
    int i, size, p_sock;
    int pdulen = sizeof(PDU);
    struct hostent *hp;         // Host entity for host information
    PDU recvPDU;                // Structure to store recieved PDUs
    struct sockaddr_in fsin;    // Store from address of clients

    int n;

    for(n=0; n<CONNECTIONSIZE; n++){ // Initialize all elements in file list to null
        file_list[n].start = NULL;
    }

    // Check number of argc and perform accordingly
    switch (argc) {
```

```
    // If 1 argument --> Do nothing
    case 1:
        break;

    // If 2 arguments --> Set 'server' to port number
    case 2:
        server = argv[1];
        break;

    // If more than 2 arguments --> print usage
    default:
        fprintf(stderr, "[-] Usage: %s [port]\n", argv[0]);
}

memset(&sin, 0, sizeof(sin));       // Sockaddr_in in initialized
sin.sin_family = AF_INET;           // Set 'sin' address to AF_NET
sin.sin_addr.s_addr = INADDR_ANY;   // Socket can bing to any available interface
sin.sin_port = htons((u_short)atoi(server)); // Map server name to port

// Allocate socket specifying UDP
int s = socket(AF_INET, SOCK_DGRAM, 0);

// Check if socket was created successfully
if (s < 0){
    fprintf(stderr, "[!] ERROR: Socket could NOT be created\n");
    exit(1);
}
fprintf(stderr,"[+] Socket created\n");


// Check if socket could be bound
if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0){
    fprintf(stderr, "[!] ERROR: Can NOT bind to %s port\n", server);
}
fprintf(stderr, "[+] Bind to Socket Successful\n");

// Main loop
while (1) {

    // Recieve PDU (recvPDU) using 's' socket
    if ((n=recvfrom(s, &recvPDU, pdulen, 0,(struct sockaddr *)&fsin, &alen)) < 0){
        printf("[!] ERROR: Recieve PDU Error n=%d\n",n);
    }

    // If PDU type = R (Registration)
    if(recvPDU.PDU_type == 'R'){
        // Handle Registration Request
        registration(s, recvPDU.PDU_data, &fsin);
        continue;
    }

    // If PDU type = S (Search Content)
    if(recvPDU.PDU_type == 'S'){
        // Handle Search for Content
        search(s, recvPDU.PDU_data, &fsin);
        continue;
    }

    // If PDU type = O (List of On-Line Registered Content)
    if(recvPDU.PDU_type == 'O'){
        printf("[O] Request to list all file content\n");
        size = 0;
        tpdu.PDU_type = 'O';    // set PDU type to 'O'

        // For-Loop to gather File names
        for(n=0; n<indexMax; n++){

            // If file_list is not NULL
            if(file_list[n].start != NULL){
                // Copy the file name to the PDU data
                strcpy(&tpdu.PDU_data[size], file_list[n].name);
                size = size + strlen(file_list[n].name);
                tpdu.PDU_data[size] = '\n'; // Seperates file names with /n
                size = size+1;
            }
        }

        tpdu.PDU_data[size] = '\0'; // Add null terminator to the end of PDU data
        sendto(s, &tpdu, sizeof(tpdu), 0,(struct sockaddr *) &fsin, sizeof(fsin)); // Send response PDU to peer
        continue;
    }

    // If PDU type = T (Deregistration)
    if(recvPDU.PDU_type == 'T'){
        // Handle De-Registration Request
        deregistration(s, recvPDU.PDU_data, &fsin);
        continue;
    }
```

```
        printf("[!] ERROR: with the PDU protocol\n");

    }
    return(0);
}

/* REGISTRATION METHOD */
void registration(int s, char *PDU_data, struct sockaddr_in *addr){

    // Initialize Variables
    INPUT *index, *temp;
    int n, duplicate = -1, located= -1;                 // Flags for duplicates and location
    int p_sock;                                         // Socket Descriptor
    char name[FILENAMESIZE], user[FILENAMESIZE];        // Arrays for File and User Names
    struct sockaddr_in p_addr;                          // To Store Peer address
    PDU tpdu;

    memcpy(&p_addr, addr, sizeof(struct sockaddr)); // Copy peer adress
    strcpy(name, &PDU_data[FILENAMESIZE]);          // Copy file anme from recvPDU data
    strcpy(user, PDU_data);                         // Copy username from recvPDU data
    memcpy(&p_addr.sin_port, &PDU_data[FILENAMESIZE+FILENAMESIZE], sizeof(p_addr.sin_port));    // Copy peer port num from recvPDU data
    temp = (INPUT *) malloc(sizeof(INPUT));         // Allocate memory for a new linked list node

    memcpy(&temp->addr, &p_addr, sizeof(struct sockaddr_in));   // Copy peer address to new node
    strcpy(temp->user, user);                                   // Copy user name to new node
    temp->next = NULL;                                          // Set next point to null

    // Search for record in 'file_list' array
    for(n=0; n<indexMax; n++){
        // Check if content name matches
        if((strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL)){
            located = n; // Record location
            break;
        }
    }

    // If file is not registered already
    if(located == -1){  // First peer that registers the file
        // Find empty spot in 'file_list'
        for(n=0; n<=indexMax; n++){
            // Check if current slot is empty
            if(file_list[n].start==NULL){
                file_list[n].start = temp;          // Set starting node of linked list to temp
                strcpy(file_list[n].name, name);    // Copy file name to the 'file_list' array
                temp->token = 1;                    // New node token = 1

                if(n == indexMax)
                    ++indexMax;

                // Set PDU type - A (Acknoledgement)
                tpdu.PDU_type = 'A';
                printf("[R] Registration: File %s registered:\n",name);
                printf("[R] Registration: Peer name: %s\n",user);
                printf("[R] Registration: %s IP Address %s\n",user, inet_ntoa(p_addr.sin_addr));
                printf("[R] Registration: %s  port number %d\n",user, ntohs(p_addr.sin_port));
                break;
            }
        }

        // Send PDU Acknoledgement to client
        sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));

    }else{
        index = file_list[located].start;

        // Checks for duplicated user names
        while(index != NULL){
            // If username is duplicated
            if(strcmp(index->user, user) == 0){
                duplicate = 1;
                break;
            }
            else{
                index = index->next; // Move to next node
            }
        }


        // If duplicate is found
        if(duplicate == 1){
            free(temp); // Free memory allocated for temp

            // Set PDU type - E (Error)
            tpdu.PDU_type = 'E';
            printf("[!] ERROR: Registration Duplicate Username Found%s\n", user);
        }
        else{ // User name is unique
            index = file_list[located].start;
            index->token = 0;   // Set starting node token to 0

            // Set tokens for all nodes to 0
            while(index->next != NULL){
                index = index->next;
                index->token = 0;
            }
```

```
            index->next = temp;
            temp->token=1;

            // Set PDU type - A (Acknoledgement)
            tpdu.PDU_type = 'A';

            printf("[R] Registration: File %s registered:\n",name);
            printf("[R] Registration: Peer name: %s\n",user);
            printf("[R] Registration: %s IP Address %s\n",user, inet_ntoa(p_addr.sin_addr));
            printf("[R] Registration: %s  port number %d\n",user, ntohs(p_addr.sin_port));
        }
        // Send SPD response to the client
        sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
    }
    return;
}

/* SEARCH METHOD */
void search(int s, char *PDU_data, struct sockaddr_in *addr){

    // Initialize Variables
    int n,located=-1;
    char name[FILENAMESIZE];
    INPUT   *start, *index;
    PDU tpdu;

    strcpy(name, PDU_data); // Copy content name from PDU data

    // Find the file from the 'file_list' array
    for(n=0; n<indexMax; n++){
        // Check if file name matches
        if((strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL) ){
            located = n; // Record location
            break;
        }
    }

    // If File name could not be found
    if(located == -1){
        // Set PDU type - E (Error)
        tpdu.PDU_type= 'E';
        printf("[!] ERROR: Search was Unsuccessful\n");

        //Send error PDU to client
        sendto(s, &tpdu, sizeof(tpdu),0, (struct sockaddr *)addr, sizeof(struct sockaddr));
        return;
    }

    else{ // Content is found
        start = file_list[located].start;
        index = start;

        // find a peer with a token set to 1
        while(index != NULL){
            // if peer token = 1 is found
            if(index->token == 1){
                printf("[+] Search: Located peer %s at port number %d\n", index->user,ntohs(index->addr.sin_port));
                index->token = 0; // Update tokens for the next search
                if(index->next == NULL){
                    start->token=1;
                }
                else{
                    index->next->token = 1;
                }

                // Set PDU types - S (Search)
                tpdu.PDU_type = 'S';

                memcpy(tpdu.PDU_data, &index->addr, sizeof(struct sockaddr));                           // Copy peer address to the PDU data
                n = sendto(s, &tpdu, sizeof(tpdu),0, (struct sockaddr *)addr, sizeof(struct sockaddr));     // Send the PDU result to client
                //printf("Search: n= %d\n",n);
                return;
            }
        index = index->next;
        }
    }
}

/* DEREGISTRATION */
void deregistration(int s, char *PDU_data, struct sockaddr_in *addr){

    // Initialize Variables
    INPUT   *index, *start, *prev;
    int n, located = -1;
    char    name[FILENAMESIZE], user[FILENAMESIZE];
    PDU tpdu;

    //printf("Entering the deregisration routine\n");
    strcpy(name, PDU_data);                 // Copy file name from recvPDU data
    strcpy(user, &PDU_data[FILENAMESIZE]);  // Copy user name from recvPDU data

    // Find file from 'file_list' array
    for(n=0; n<indexMax; n++){
        // Check if file name matches
```

```
            if((strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL) ){
                located= n;
                break;
            }
    }

    //printf("[T] Deregisration: at Location %d\n",located);
    printf("[T] Deregistration: the user %s\n", user);

    // If the content for deregistration is not found
    if(located == -1){

        // Set PDU type - 'E'
        tpdu.PDU_type = 'E';
        // Send error PDU to client
        sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
        return;
    }
    else{
        start = index = prev = file_list[located].start;

        // Find input  for the deregistration
        while(index != NULL){
            if(strcmp(index->user,user) == 0){
                located = 1;    // input is located when equal to 1
                //printf("[T] Deregistration located the input\n");
                printf("[T] Deregistration remove the input: %d\n", ntohs(index->addr.sin_port));

                // If input that will be removed is the starting node
                if(index == start){
                    file_list[located].start = index->next;
                }
                else{
                    // Remove the input from the linekd list
                    prev->next = index->next;

                    // If the removed node has the token
                    if(index->token == 1){
                        // Update token for the next deregistration
                        if(index->next == NULL){
                            file_list[located].start->token=1;
                        }
                        else{
                            index->next->token = 1;
                        }
                    }
                }

            free(index);    // free the memory allocation fro the removed input node
            break;
            }
        prev = index;
        index = index->next;
        }
    }
    // Check in input was not located
    if(located == -1)
        // Set PDU type - 'E'
        tpdu.PDU_type = 'E';
    else{
        // Check if starting node is null and located index to the last one
        if(file_list[located].start == NULL && located == indexMax-1) indexMax--;
        // Set PDU type - 'E'
        tpdu.PDU_type='T';
    }

    // Send PDU response to the client
    sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
    return;
}

```