

Course Number	COE 768
Course Title	Computer Networks
Semester/Year	Fall 2023
Instructor	Dr. Truman Yang
TA name	Parastoo Jafarzadeh

Lab No.	Final Project
----------------	---------------

Report Title	P2P Application
---------------------	-----------------

Section No.	15
Submission Date	November 30th, 2023
Due Date	December 1st, 2023

Student Name	Student ID	Signature*
Vidy Matadeen	501054327	V.M
Maria Labeeba	500960386	M.L

Student Name Student ID Signature*

*By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission 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

Table of Content

1. Introduction	3
2. Description of the client and server programs	3
2.1 Client	3
2.2 Server	3
2.3 Protocol Data Unit Types	4
2.3.1 Content Registration [R]	4
Figure 1	4
Figure 2	4
2.3.2 Content Download [D]	5
Figure 3	6
Figure 4	7
Figure 5	8
2.3.3 Online-Listing of Content [O]	8
Figure 6	8
2.3.4 Content Deregistration [T]	8
Figure 7	9
2.3.5 Quit [Q]	9
Figure 8	9
Table 1	10
3. Observations and analysis	10
Figure 9	10
Figure 10	11
Figure 11	11
Figure 12	12
Figure 13	12
Figure 14	13
Figure 15	13
Figure 16	14
Figure 17	14
Figure 18	15
4. Conclusions	15
5. Appendix	16
5.1 Text File Contents	16
5.2 Peer.c Code	17
5.3 Server.c Code	23

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.

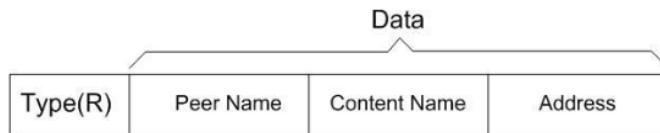


Figure 1: R Type Data Packet

```

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'){
276             FD_SET(p_sock, &afds);                                // Check if the received PDU type is an Acknowledgement
277             table[p_sock].status = 1;                             // Listening on TCP socket
278             strcpy(table[p_sock].name,name);                      // Set status for registration to 1
279
280             if(nfds <= p_sock){                                 // Copy filename to table
281                 nfds = p_sock+1;                            // Updates the max file descriptor 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",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 successfully 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 byte 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 successfully
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 = '0';           // Set PDU type to '0'
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     }

```

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 check 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     }
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
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

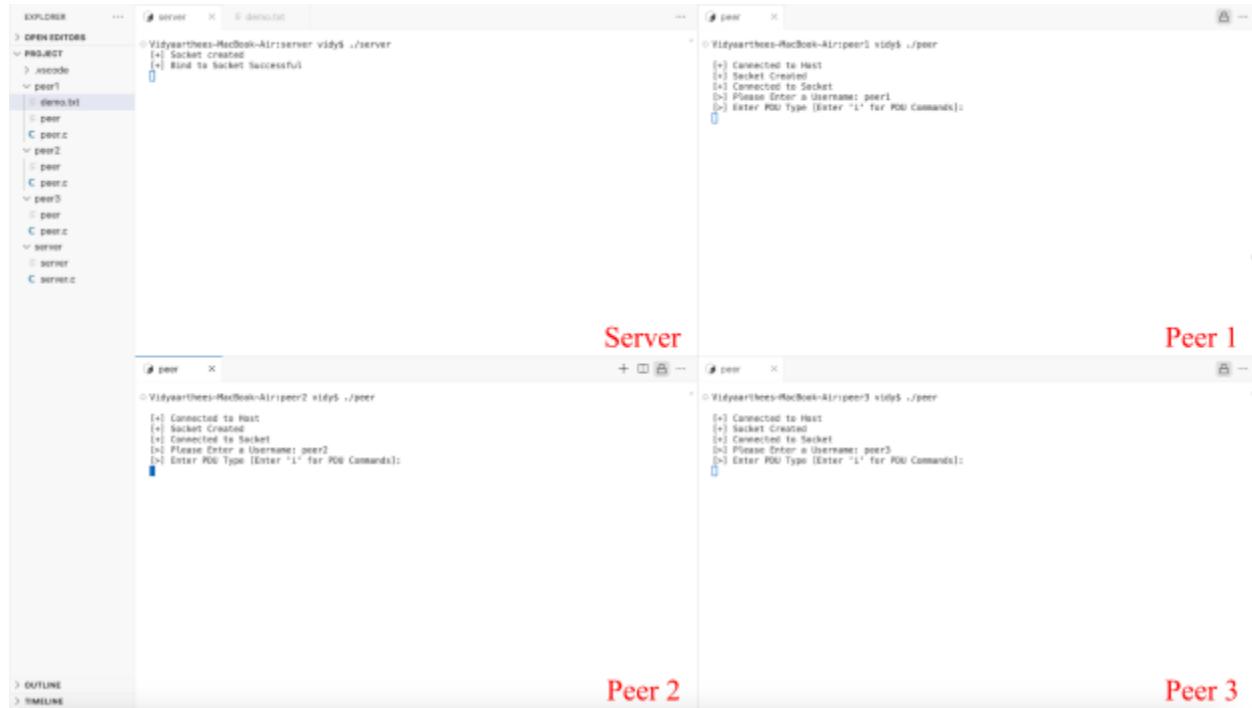


Figure 9: Running the three peers and the index server

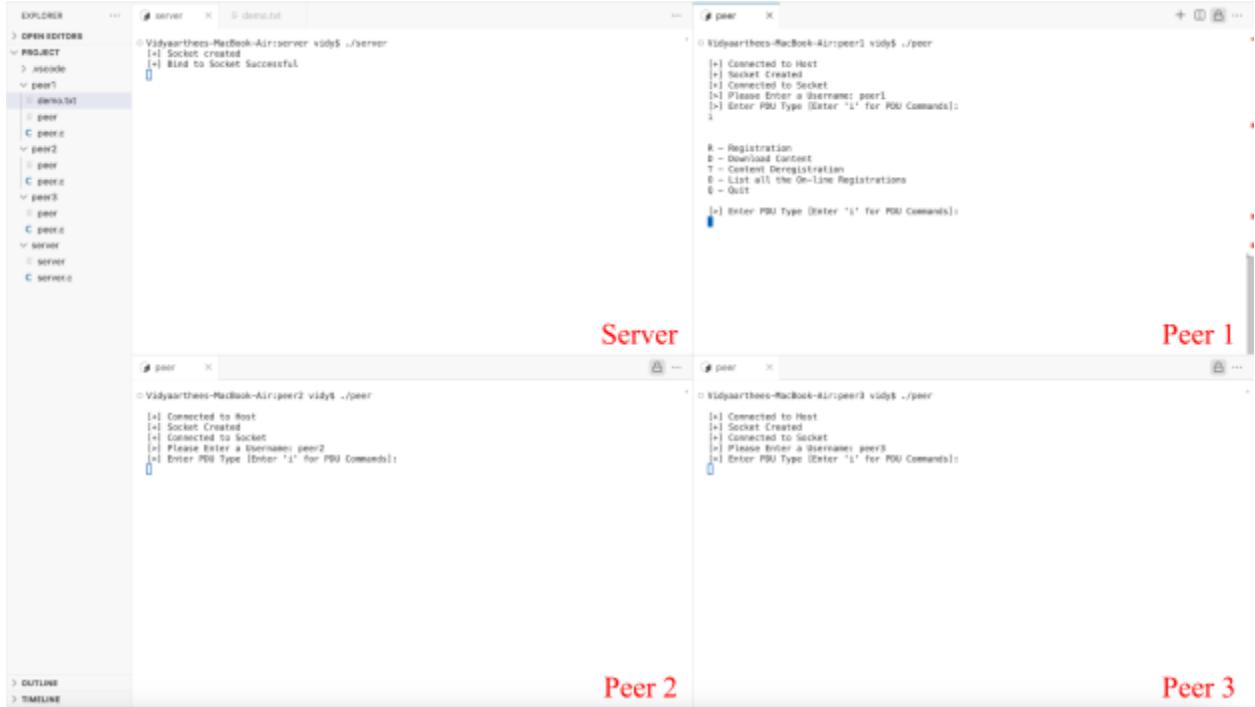


Figure 10: Peer 1 enters the 'i' command to display the menu of possible commands to enter into the terminal.



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.

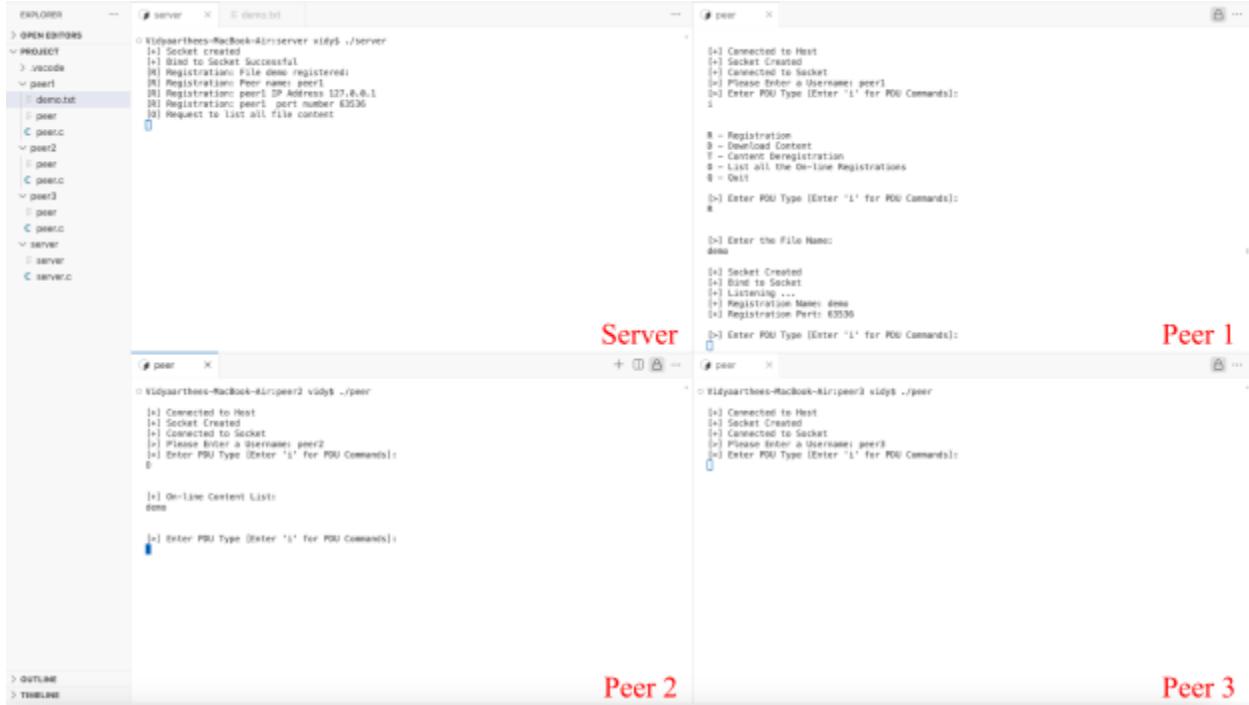


Figure 12: Peer 2 requests a list of the contents registered to the index server. The server displays a prompt stating a request was made. The peer receives a list of the files that have been registered. In this example we can see the demo.txt file is registered.



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.

The screenshot shows three terminal windows labeled Server, Peer 1, and Peer 2.

- Server:**

```
Vidyaarthi-MacBook-Air:server vidy$ ./server
[+] Socket created
[+] Bind to Secker Successful
[+] Registration: File demo registered:
[+] Registration: Peer name: peer1
[+] Registration: Peer IP Address 127.0.0.1
[+] Registration: Peer Port number 63536
```
- Peer 1:**

```
Vidyaarthi-MacBook-Air:peer1 vidy$ ./peer
[+] Connected to Host
[+] Socket Created
[+] Connected to Socket
[+] Please Enter a Username: peer1
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
[+] Registration
[+] Download Content
[+] Content Deregistration
[+] List all the On-line Registrations
[+] Quit
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
R
[+] Enter the File Name:
demo
[+] Socket Created
[+] Bind to Socket
[+] Listening ...
[+] Registration Name: demo
[+] Registration Port: 63536
```
- Peer 2:**

```
Vidyaarthi-MacBook-Air:peer2 vidy$ ./peer
[+] Connected to Host
[+] Socket Created
[+] Connected to Socket
[+] Please Enter a Username: peer2
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
[+] On-line Content List:
demo
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
B
[+] Enter the File Name you wish to Download:
demo
[+] Search results successfully found
[+] Download from Peer IP Address at: 167773043
[+] Download from Peer Port Number: 63536
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
```

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

The screenshot shows three terminal windows labeled Server, Peer 1, and Peer 2.

- Server:**

```
Vidyaarthi-MacBook-Air:server vidy$ ./server
[+] Socket created
[+] Bind to Secker Successful
[+] Registration: File demo registered:
[+] Registration: Peer name: peer1
[+] Registration: Peer IP Address 127.0.0.1
[+] Registration: Peer Port number 63536
[+] Request to list all file content
[+] Search: Located peer1 peer1 at port number 63536
[+] Deregistration: the user peer1
[+] Deregistration remove the input: 63536
```
- Peer 1:**

```
Vidyaarthi-MacBook-Air:peer1 vidy$ ./peer
[+] Please Enter a Username: peer1
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
[+] Registration
[+] Download Content
[+] Content Deregistration
[+] List all the On-line Registrations
[+] Quit
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
R
[+] Enter the File Name:
demo
[+] Socket Created
[+] Bind to Socket
[+] Listening ...
[+] Registration Name: demo
[+] Registration Port: 63536
```
- Peer 2:**

```
Vidyaarthi-MacBook-Air:peer2 vidy$ ./peer
[+] Connected to Host
[+] Socket Created
[+] Connected to Socket
[+] Please Enter a Username: peer2
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
[+] On-line Content List:
demo
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
B
[+] Enter the File Name you wish to Download:
demo
[+] Search results successfully found
[+] Download from Peer IP Address at: 167773043
[+] Download from Peer Port Number: 63536
[+] Enter PDU Type [Enter 'l' for PDU Commands]:
l
```

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.

The screenshot shows three terminal windows labeled Server, Peer 1, and Peer 2.

- Server:**
 - Shows the file structure: .server, .demo.txt, .peer, .peer1, .peer2, .peer3, .server, .service.
 - Terminal output for .server command:
 - [+] Socket created
 - [+] Bind to Socket Successful
 - [+] Peer1 Connected to Host
 - [+] Peer1 Registered
 - [+] Registration: Peer Name: peer1
 - [+] Registration: peer1 IP Address: 127.0.0.1
 - [+] Registration: peer1 port number 63536
 - [+] Search: Located peer peer1 at port number 63538
 - [+] Request to list all file content
 - [+] De-registration: the user peer1
 - [+] De-registration remove the input: 63536
 - [+] Request to list all file content
- Peer 1:**
 - Terminal output for .boot command:
 - [+] Please Enter a Username: peer1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: R
 - [+] Enter the File Name: demo
 - [+] Socket Created
 - [+] Bind to Socket
 - [+] Listening ...
 - [+] Registration Name: demo
 - [+] Registration Port: 63538
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] De-Registration Content demo is successfully de-registered
- Peer 2:**
 - Terminal output for .boot command:
 - [+] Please Enter a Username: peer2
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] On-Line Content List: demo
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter the File Name you wish to Download: demo
 - [+] Search results successfully found
 - [+] Download from Peer IP Address: 127.0.0.1
 - [+] Download from Peer Port Number: 63538
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0

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.

The screenshot shows three terminal windows labeled Server, Peer 1, and Peer 2.

- Server:**
 - Shows the file structure: .server, .demo.txt, .peer, .peer1, .peer2, .peer3, .server, .service.
 - Terminal output for .server command:
 - [+] Socket created
 - [+] Bind to Socket Successful
 - [+] Peer1 Connected to Host
 - [+] Peer1 Registered
 - [+] Registration: Peer Name: peer1
 - [+] Registration: peer1 IP Address: 127.0.0.1
 - [+] Registration: peer1 port number 63536
 - [+] Search: Located peer peer1 at port number 63538
 - [+] Request to list all file content
 - [+] De-registration: the user peer1
 - [+] De-registration remove the input: 63536
 - [+] Request to list all file content
 - [+] De-registration: the user peer1
 - [+] De-registration remove the input: 63538
- Peer 1:**
 - Terminal output for .boot command:
 - [+] Please Enter a Username: peer1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: R
 - [+] Enter the File Name: demo
 - [+] Socket Created
 - [+] Bind to Socket
 - [+] Listening ...
 - [+] Registration Name: demo
 - [+] Registration Port: 63538
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] De-Registration Content demo is successfully de-registered
- Peer 2:**
 - Terminal output for .boot command:
 - [+] Please Enter a Username: peer2
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 1
 - [+] On-Line Content List: demo
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] Enter the File Name you wish to Download: demo
 - [+] Search results successfully found
 - [+] Download from Peer IP Address: 127.0.0.1
 - [+] Download from Peer Port Number: 63538
 - [+] Enter PDU Type [Enter '1' for PDU Commands]: 0
 - [+] De-Registration content demo is successfully de-registered

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

```

server
Vidyarthi-MacBook-Air:~/server vidysh$ ./server
[+] Socket created
[+] Bind to Socket Successful
[+] Registration: File demo registered
[+] Registration: Peer peer1 registered
[+] Registration: Peer IP Address: 127.0.0.1
[+] Registration: peer1 port number 63536
[+] Request to list all file contents
[+] Search: Located peer peer1 at port number 63536
[+] Request to list all file content
[+] Deregistration: The user peer1
[+] Registration: Peer peer2 registered
[+] Registration: Peer IP Address: 127.0.0.1
[+] Registration: peer2 port number 63536
[+] Request to list all file content
[+] Deregistration: The user peer2
[+] Deregistration remove the input: 63528
[+] Request to list all file content

Server
Vidyarthi-MacBook-Air:~/peer vidysh$ ./peer
[+] Connected to Host
[+] Socket Created
[+] Connected to Socket
[+] Please Enter a Username: peer3
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] On-Line Content List:
demo

[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter the File Name you wish to Download: demo
[+] Search results successfully found
[+] Download From Peer IP Address: 127.0.0.1
[+] Download From Peer Port Number: 63536
[+] ERROR: Download content file unavailable for remote peer
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] De-Registration Content demo is successfully de-registered
Vidyarthi-MacBook-Air:~/peer vidysh$ 

Peer 1
Please Enter a Username: peer3
Enter PDU Type [Enter '1' for PDU Commands]: 0
M = Registration
D = Download Content
R = Content De-registration
L = List all the On-line Registrations
Q = Quit

Enter PDU Type [Enter '1' for PDU Commands]: R
R = 

Enter the File Name: demo
[+] Socket Created
[+] Bind to Socket
[+] Listening ...
[+] Registration Name: demo
[+] Registration Port: 63536

Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] De-Registration Content demo is successfully de-registered

Peer 2
[+] Connected to Host
[+] Socket Created
[+] Connected to Socket
[+] Please Enter a Username: peer3
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] On-Line Content List:
demo

[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] On-Line Content List:
demo

[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] On-Line Content List:
demo

[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] On-Line Content List:
demo

Peer 3
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0
[+] Enter PDU Type [Enter '1' for PDU Commands]: 0

```

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

```
demo.txt X
peer1 > demo.txt
1 This is our text file that we will be using for
2 demo purposes. We will start with it in peer1.
3 Then it will be registered and downloaded to
4 peer2. Peer2 will automatically be registered as
5 a content server for the text file. At this point
6 peer3 will download this file and register as a
7 content server for this content.
8
9 Test File For P2P Project testing : ) 371 bytes
```

5.2 Peer.c Code

```

1  /* COE 768 - Final Project
2   -----
3   Name(s): Maria Labbeba (500960386)
4   & Vidy Matadeen (501054327)
5   Due Date: Nov 27th, 2023
6   File: peer.c
7
8 */
9
10  /* PDU PDU_type      Function          Direction
11   -----
12   R    Content Registration        Peer to Index Server
13   D    Content Download Request  Content Client to Content Server
14   S    Search for content and the  Between Peer and Index Server
15   |    |    |    associated content server
16   T    Content De-Registration    Peer to Index Server
17   C    Content Data             Content Server to Client
18   O    List of On-Line Registered Between Peer and Index Server
19   |    |    |    Content
20   A    Acknowledgement         Index Server to Peer
21   E    Error                   Between Peers or between Peer and
22   |    |    |    Index Server
23 */
24
25  /* IMPORTS */
26  #include <stdio.h>
27  #include <netdb.h>
28  #include <sys/types.h>
29  #include <sys/socket.h>
30  #include <netinet/in.h>
31  #include <stdlib.h>
32  #include <strings.h>
33  #include <string.h>
34  #include <arpa/inet.h>
35  #include <unistd.h>
36  #include <sys/stat.h>
37  #include <fcntl.h>
38  #include <sys/select.h>
39  #include <errno.h>
40
41  /* DEFINITIONS */
42  #define QUIT "quit"                // Define string 'quit' for quitting program
43  #define SERVER_PORT 15000           // Define Port of server
44  #define BUflen 100                 // Define length of PDU_data
45  #define FILENAMESIZE 10            // Define max size for file name
46  #define CONNECTIONSIZE 100          // Define max number of connections
47
48  /* STRUCT: PDU */
49  typedef struct {
50      char PDU_type;              // Character to represent PDU type
51      char PDU_data[BUflen];       // Array stores data in PDU
52  } PDU;
53  PDU recvPDU;                  // Received PDU
54
55  struct {
56      int status;                // Integer to represent the status bit of file
57      char name[FILENAMESIZE];    // Array stores filename connected
58  } table[CONNECTIONSIZE];       // Keeps track of registered content
59
60  char user[FILENAMESIZE];       // Array stores user name with max size of FILENAME
61
62  /* DESCRIPTORS */
63  int s_sock;                    // Socket Descriptor
64  int fd;                        // File Descriptor
65  int nfds;                      // Max Number of Files Descriptor
66  fd_set rdfs, afds;             // Setting Reading and Active File Descriptors
67
68  /* FUNCTIONS */
69  void registration(int, char *);  // Content Registration
70  int search_content(int, char *, PDU *); // Content Search
71  int client_download(char *, PDU *); // Content Download by Client
72  void server_download();          // Content Download by Server
73  void deregistration(int, char *); // Content Deregistration
74  void listing(int);              // Listing Online Content
75  void quit(int);                // Quitting Program
76
77  /* Main Functions */
78  int main(int argc, char **argv){
79      int s_port = SERVER_PORT;     // Default server port 15000
80      int n;                      // Used in For - Loops
81      int alien = sizeof(struct sockaddr_in); // Length of Socket Address
82      struct hostent *hp;          // Host input pointer
83      struct sockaddr_in server;   // ServSocket address structure
84      char typePDU, name[FILENAMESIZE]; // For user input
85      char *host = "localhost";    // Default host name
86      struct sigaction sa;         // Signal Action Structure
87
88

```

```

89     /* SWITCH CASE FOR COMMAND LINE HANDLING*/
90     switch(argc){
91     case 1:
92         break;
93     case 2:
94         host = argv[1];
95         break;
96     case 3:
97         host = argv[1];
98         s_port = atoi(argv[2]);
99         break;
100    default:
101        printf(stderr, "[-] Usage: %s host [port]\n", argv[0]);
102        exit(1);
103    }
104
105    /* UDP Connection with the index server */
106    memset(&server, 0, alen);           // Server address sockets set to 0
107    server.sin_family = AF_INET;
108    server.sin_port = htons(s_port);
109
110    /* Establishing Host Connection */
111    if ( hp = gethostbyname(host) )
112        memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
113    else if ( (server.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE ){
114        printf("\n[!] Can NOT connect to host \n");
115        exit(1);
116    }
117    printf("\n[+] Connected to Host\n");
118
119    /* Establishing Socket Connection */
120    s_sock = socket(PF_INET, SOCK_DGRAM, 0);
121    if (s_sock < 0){
122        printf("[!] Can NOT create Socket\n");
123        exit(1);
124    }
125    printf("[+] Socket Created\n");
126
127    if (connect(s_sock, (struct sockaddr *)&server, sizeof(server)) < 0){
128        printf("[!] Can NOT connect to Socket\n");
129        exit(1);
130    }
131    printf("[+] Connected to Socket\n");
132
133    /* User Name Configuration */
134    printf("[>] Please Enter a Username: ");
135    scanf("%s", user);
136
137    /* Initialization of SELECT structure and table structure */
138    FD_ZERO(&rfds);
139    FD_SET(s_sock, &rfds);           // Listening on TCP socket
140    FD_SET(0, &rfds);             // Listening on stdin
141    nfds = 1;
142
143    for(n=0; n<CONNECTIONSIZE; n++)      // Set all status values to -1
144        table[n].status = -1;
145
146
147    /* Main Loop */
148    while(1{
149
150        printf("[>] Enter PDU Type [Enter 'i' for PDU Commands]:\n ");
151        memcpy(&rfds, &fd, sizeof(rfds));
152
153
154        // If there is not input, return Error
155        if (select(nfds, &rfds, NULL, NULL, NULL) == -1){
156            printf("[!] Error: %s\n", strerror(errno));
157            exit(1);
158        }
159
160        // If there is an input, read character as PDU Type from user
161        if (FD_ISSET(0, &rfds)) {
162            typePDU = getchar();
163
164            /* PDU Type Command Options */
165            if(typePDU == 'i'){
166                printf("\nR - Registration\n");
167                printf("D - Download Content\n");
168                printf("T - Content Deregistration\n");
169                printf("L - List all the On-line Registrations\n");
170                printf("Q - Quit\n\n");
171                continue;
172            }
173        }

```

```

174     /* PDU Type R - Registration */
175     if(typePDU == 'R') {
176         printf("\n[+] Enter the File Name:\n");
177         scanf("%s",name); //Filename input
178         registration(s_sock, name); //name passed to registration function
179         continue;
180     }
181
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     }
193
194     /* PDU Type O - List on-line Content */
195     if(typePDU == 'O'){
196         listing(s_sock);
197         continue;
198     }
199
200     /* PDU Type 'T' - Content Deregistration */
201     if(typePDU == 'T'){
202         printf("\n[+] Enter the Filename you wish to Deregister:\n");
203         scanf("%s",name);
204         deregistration(s_sock, name);
205         continue;
206     }
207
208     /* PDU Type 'Q' - Quit */
209     if(typePDU == 'Q'){
210         quit(s_sock);
211         exit(1);
212     }
213
214     /* Server to client File Transfer*/
215     server_download(s_sock);
216 }
217 return 0;
218 }

219 /* REGISTRATION METHOD */
220 void registration(int s_sock,char *name){
221
222     // Initialize variables
223     struct sockaddr_in reg_addr, addr; // Socket address structure
224     int p_sock, alen, i; // Peer socket descriptor, socket length address struct
225     int p_port = 0; // peer port
226     PDU tpdu, recvPDU; // PDU structure for sending and recieving data
227
228     alen = sizeof(struct sockaddr_in); // Gets the size of the socket
229
230
231     // Create a Peer socket
232     if ((p_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
233         printf("[!] ERROR: Registration Can NOT create socket\n");
234         return;
235     }
236     printf("\n[+] Socket Created\n");
237
238     // Bind to Peer Socket
239     memset(&reg_addr, 0, sizeof(struct sockaddr_in)); // Initialize sockaddr_in
240     reg_addr.sin_family = AF_INET; // Set address to AF_INET
241     reg_addr.sin_port = htons(0); // Set the port to 0
242     reg_addr.sin_addr.s_addr = htonl(INADDR_ANY); // Allows socket to bind to available local IP address
243
244     if (bind(p_sock, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) == -1){
245         printf("[!] ERROR: Registration Can NOT bind to socket\n");
246         return;
247     }
248     printf("[+] Bind to Socket\n"); // Content Server Bound
249
250     listen(p_sock,5);
251     printf("[+] Listening ... \n");
252
253     getsockname(p_sock, (struct sockaddr *)&reg_addr, &alen); //retrieves the current name for the specified p socket descriptor
254
255     tpdu.PDU_type = 'R'; // Set PDU Type 'R' for registration
256     memcpy(&tpdu.PDU_data[FILENAMESIZE], name, FILENAMESIZE); // Copy content name to PDU data
257     memcpy(&tpdu.PDU_data[2*FILENAMESIZE],&reg_addr.sin_port,sizeof(reg_addr.sin_port)); // Copy port to PDU data
258
259     while(1){
260         memcpy(tpdu.PDU_data, user, FILENAMESIZE); // Copy user to PDU data
261
262         // Error Message
263         if (iowrite(s_sock, &tpdu, sizeof(tpdu)) <=0 ){
264             printf("[!] ERROR: Registration Write Error\n");
265             return;
266         }

```

```

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'){
276     FD_SET(p_sock, &nfds);
277     table[p_sock].status = 1;
278     strcpy(table[p_sock].name,name);
279
280     if(nfds <= p_sock){
281         nfds = p_sock+1;
282     }
283
284     printf("[+] Registration Name: %s\n",name);
285     printf("[+] Registration Port: %d\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';
305     memcpy(tpdu.PDU_data, name, FILENAMESIZE);
306
307     // Error Messages
308     if((n=write(s_sock, &tpdu, BUFLEN+1)) < 0){
309         printf("(!) ERROR: Search Content Write Error\n");
310         return -1;
311     }
312
313     if((n = read(s_sock, recvPDU, BUFLEN+1))<0){
314         printf("(!) ERROR: Search Content Read Error\n");
315         return -1;
316     }
317
318     if(recvPDU->PDU_type == 'E'){
319         // Check If the recvPDU type is E Indicating an Error
320         printf("(!) ERROR: %s - Filename can not be Found \n",name);
321         return -1;
322     }
323
324     if(recvPDU->PDU_type == 'S'){
325         // Check If the type is S indicating Search success
326         printf("\n[+] Search results successfully found\n");
327         return 0;
328     }
329     else{
330         printf("(!) ERROR: Search Content Protocol Error\n");
331         return -1;
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));
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 }
```

```

358     tpdu.PDU_type = 'D';           // Set PDU type to 'D'
359     write(p_sock, &tpdu, 1);      // Writing 1 byte 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 successfully
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 isn't available 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;
389 }
390
391 /* LISTING METHOD */
392 void listing(int s_sock){
393
394     // Initialize Variables
395     PDU tpdu;
396     tpdu.PDU_type = '0';           // Set PDU type to '0'
397
398     write(s_sock, &tpdu, sizeof(tpdu)); // writing from the pointer tpdu to the server socket
399     read(s_sock, &recvPDU, sizeof(recvPDU)); // Read response received PDU from the server socket
400
401     printf("\n[+] On-line Content List:\n%s\n", recvPDU.PDU_data);
402     return;
403 }
404
405 /* DEREGRIGATION METHOD */
406 void deregistration(int s_sock,char *name){
407
408     // Initialize Variables
409     int i;
410     int located = -1;
411     PDU tpdu, recvPDU;
412
413     // Checks and updates to see if the file is registered
414     for(i=3; i<nfds; i++){
415         if(strcmp(table[i].name, name)==0){
416             if(table[i].status == 1){
417                 located = i;
418             }
419         }
420     }
421
422     // If content was not found or registered
423     if(located == -1){
424         printf("[-] ERROR: De-Registration Content %s is not registered\n", name);
425         return;
426     }
427     else{
428         // If content was already registered
429         if(table[located].status == -1){
430             printf("[-] ERROR: De-Registration Content %s is not registered\n", name);
431             return;
432         }
433
434         FD_CLR(located, &afds); // Clear file disruptors
435         if(nfds == located+1) nfds = located; // Update Max file disruptors when necessary
436         table[located].status = -1; // If located we reset the status bit
437         close(located);
438     }
439
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 if 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 }
456
457 /* QUIT METHOD */
458 void quit(int s_sock)
{
459     int i;
460     for(i=3; i<nfds; i++){
461         if(table[i].status == 1)
462             deregistration(s_sock,table[i].name);
463     }
464     return;
465 }
466
467 /* SERVER DOWNLOAD METHOD*/
468 void server_download(){
469
470     // Initialize Variables
471     int n, i, new_sd, alen;
472     struct sockaddr_in client;
473     PDU tpdu;
474
475     for(i=3; i<nfds; ++i){
476         if (FD_ISSET(i, &fdsets)) {
477             new_sd = accept(i, (struct sockaddr *)&client, &alen);
478             n=read(new_sd, &recvPDU, sizeof(recvPDU));
479
480             if(recvPDU.PDU_type=='D'){
481                 fd = open(table[i].name,O_RDONLY);           // Check if the PDU type is 'D'
482                                         // open file in read only mode using the file table
483
484                 if(fd >= 0){                                // Check if the file opens successfully
485                     tpdu.PDU_type = 'C';                   // Set the PDU type to 'C'
486                     n = read(fd, tpdu.PDU_data, BUFSIZE); // read the data from tpdu data to the file in 100 byte size
487                     write(new_sd, &tpdu, n+1);            // write to the new socket from tpdu pointer
488                     while((n = read(fd, tpdu.PDU_data, BUFSIZE))>0){ // While loop handles files larger than 100 Bytes
489                         write(new_sd, tpdu.PDU_data, n);
490                     }
491                     printf("[+] Transmission Completed \n");
492                 }
493                 else{
494                     tpdu.PDU_type = 'E';           // Check if file did not open and set the PDU type to E
495                     write(new_sd, &tpdu, sizeof(tpdu)); // Write to new socket the pointer from tpdu
496                 }
497
498                 close(fd);                      // Close the read only file
499                 close(new_sd);                // Close the socket connection
500             }
501         }
502     }
503 }
504
505 }
```

5.3 Server.c Code

```

1  /* COE 768 - Final Project
2   |
3   | Name(s): Maria Labeeba (500960386)
4   |           & Vidy Matadeen (501054327)
5   | Due Date: Nov 27th, 2023
6   | File: server.c
7   |
8   */
9
10 /* PDU PDU_type          Function          Direction
11   -----          -----
12   R             Content Registration    Peer to Index Server
13   D             Content Download Request Content Client to Content Server
14   S             Search for content and the Between Peer and Index Server
15   |             associated content server
16   T             Content De-Registration Peer to Index Server
17   C             Content Data          Content Server to Content Client
18   O             List of On-Line Registered Between Peer and Index Server
19   |             Content
20   A             Acknowledgement     Index Server to Peer
21   E             Error              Between Peers or between Peer and
22   |             Index Server
23
24 /* LIBRARIES*/
25 #include <sys/types.h>
26 #include <sys/socket.h>
27 #include <netinet/in.h>
28 #include <stdlib.h>
29 #include <string.h>
30 #include <netdb.h>
31 #include <stdio.h>
32 #include <rpa/inet.h>
33
34 /* DEFINITIONS */
35 #define BUFSIZE 100      // Defines length of message buffers
36 #define FILENAMESIZE 10  // Defines Max size of file names
37 #define CONNECTIONSIZE 100 // Defines mac number of connections
38
39 /* STRUCTURE - PEER/CONTENT INPUT INFO */
40 typedef struct input{
41   char user[FILENAMESIZE]; // Array to store usernames
42   struct sockaddr_in addr; // Structure to store the socket address info
43   short token;           // Placeholder for input
44   struct input *next;    // Pointer to the next input
45 } INPUT;
46
47 /* STRUCTURE - FILE LIST */
48 typedef struct{
49   char name[FILENAMESIZE]; // Array to store name of file_list
50   INPUT *start;           // Pointer to the files
51 } FILELIST;
52
53 FILELIST file_list[CONNECTIONSIZE]; // Array with size of "CONNECTIONSIZE"
54 int indexMax=0;
55
56 /* STRUCTURE - REPRESENTS PDU */
57 typedef struct{
58   char PDU_type;        // Type of PDU
59   char PDU_data[BUFSIZE]; // Array to store PDU data
60 } PDU;
61
62 PDU tpdu;                // initialize tpdu with PDU structure
63
64 /* FUNCTION PROTOTYPES FOR SEARCH, REGISTRATION, DEREGISTRATION */
65
66 void registration(int, char *, struct sockaddr_in *);
67 void search(int, char *, struct sockaddr_in *);
68 void deregistration(int, char *, struct sockaddr_in *);
69
70
71 /* MAIN METHOD */
72 int main(int argc, char *argv[]){
73
74   // Initialize variables
75   struct sockaddr_in sin, *p_addr;
76   char *server = "15000";
77   char name[FILENAMESIZE], user[FILENAMESIZE];
78   int alen = sizeof(struct sockaddr_in);
79   int i, size, p_sock;
80   int pdulen = sizeof(PDU);
81   struct hostent *hp;           // Host entity for host information
82   PDU recvPDU;                // Structure to store received PDUs
83   struct sockaddr_in fsin;     // Store from address of clients
84
85   int n;
86
87   for(n=0; n<CONNECTIONSIZE; n++){ // Initialize all elements in file list to null
88     file_list[n].start = NULL;
89   }
90
91   // Check number of argc and perform accordingly
92   switch (argc) {
93

```

```

94 // If 1 argument --> Do nothing
95 case 1:
96 | break;
97
98 // If 2 arguments --> Set 'server' to port number
99 case 2:
100 | server = argv[1];
101 | break;
102
103 // If more than 2 arguments --> print usage
104 default:
105 | | fprintf(stderr, "[-] Usage: %s [port]\n", argv[0]);
106 }
107
108 memset(&sin, 0, sizeof(sin)); // Sockaddr_in in initialized
109 sin.sin_family = AF_INET; // Set 'sin' address to AF_NET
110 sin.sin_addr.s_addr = INADDR_ANY; // Socket can bind to any available interface
111 sin.sin_port = htons((u_short)atoi(server)); // Map server name to port
112
113 // Allocate socket specifying UDP
114 int s = socket(AF_INET, SOCK_DGRAM, 0);
115
116 // Check if socket was created successfully
117 if (s < 0){
118 | | fprintf(stderr, "[!] ERROR: Socket could NOT be created\n");
119 | | exit(1);
120 }
121 fprintf(stderr,"[+] Socket created\n");
122
123 // Check if socket could be bound
124 if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0){
125 | | fprintf(stderr, "[!] ERROR: Can NOT bind to %s port\n", server);
126 }
127 fprintf(stderr, "[+] Bind to Socket Successful\n");
128
129 // Main loop
130 while (1) {
131
132 // Receive PDU (recvPDU) using 's' socket
133 if ((n=recvfrom(s, &recvPDU, pdulen, 0,(struct sockaddr *)&fsin, &alen)) < 0){
134 | | printf("[!] ERROR: Recieve PDU Error n=%d\n",n);
135 }
136
137
138 // If PDU type = R (Registration)
139 if(recvPDU.PDU_type == 'R'){
140 | | // Handle Registration Request
141 | | registration(s, recvPDU.PDU_data, &fsin);
142 | | continue;
143 }
144
145 // If PDU type = S (Search Content)
146 if(recvPDU.PDU_type == 'S'){
147 | | // Handle Search for Content
148 | | search(s, recvPDU.PDU_data, &fsin);
149 | | continue;
150 }
151
152 // If PDU type = O (List of On-Line Registered Content)
153 if(recvPDU.PDU_type == 'O'){
154 | | printf("[0] Request to list all file content\n");
155 | | size = 0;
156 | | t pdu.PDU_type = 'O'; // set PDU type to 'O'
157
158 // For-Loop to gather File names
159 for(n=0; n<indexMax; n++){
160
161 | | // If file_list is not NULL
162 | | if(file_list[n].start != NULL){
163 | | | | // Copy the file name to the PDU data
164 | | | | strcpy(&t pdu.PDU_data[size], file_list[n].name);
165 | | | | size = size + strlen(file_list[n].name);
166 | | | | t pdu.PDU_data[size] = '\n'; // Separates file names with /n
167 | | | | size = size+1;
168 | | }
169 }
170
171 | | t pdu.PDU_data[size] = '\0'; // Add null terminator to the end of PDU data
172 | | sendto(s, &t pdu, sizeof(t pdu), 0,(struct sockaddr *)&fsin, sizeof(fsin)); // Send response PDU to peer
173 | | continue;
174 }
175
176 // If PDU type = T (Deregistration)
177 if(recvPDU.PDU_type == 'T'){
178 | | // Handle De-Registration Request
179 | | deregistration(s, recvPDU.PDU_data, &fsin);
180 | | continue;
181 }
182

```

```

183     printf("(!) ERROR: with the PDU protocol\n");
184 }
185 return(0);
186 }

/* REGISTRATION METHOD */
189 void registration(int s, char *PDU_data, struct sockaddr_in *addr){
190
191     // Initialize Variables
192     INPUT *index, *temp;
193     int n, duplicate = -1, located= -1;           // Flags for duplicates and location
194     int p_sock;                                // Socket Descriptor
195     char name[FILENAMESIZE], user[FILENAMESIZE]; // Arrays for File and User Names
196     struct sockaddr_in p_addr;                  // To Store Peer address
197     PDU tpdu;
198
199     memcpy(&p_addr, addr, sizeof(struct sockaddr)); // Copy peer address
200     strcpy(name, &PDU_data[FILENAMESIZE]);          // Copy file name from recvPDU data
201     strcpy(user, PDU_data);                        // Copy username from recvPDU data
202     memcpy(&p_addr.sin_port, &PDU_data[FILENAMESIZE+FILENAMESIZE], sizeof(p_addr.sin_port)); // Copy peer port num from recvPDU data
203     temp = (INPUT *) malloc(sizeof(INPUT));        // Allocate memory for a new linked list node
204
205     memcpy(&temp->addr, &p_addr, sizeof(struct sockaddr_in)); // Copy peer address to new node
206     strcpy(temp->user, user);                      // Copy user name to new node
207     temp->next = NULL;                            // Set next point to null
208
209     // Search for record in 'file_list' array
210     for(n=0; n<indexMax; n++){
211         // Check if content name matches
212         if(strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL)){
213             located = n; // Record location
214             break;
215         }
216     }
217
218
219     // If file is not registered already
220     if(located == -1){ // First peer that registers the file
221         // Find empty spot in 'file_list'
222         for(n=0; n<indexMax; n++){
223             // Check if current slot is empty
224             if(file_list[n].start==NULL){
225                 file_list[n].start = temp; // Set starting node of linked list to temp
226                 strcpy(file_list[n].name, name); // Copy file name to the 'file_list' array
227                 temp->token = 1;                // New node token = 1
228
229                 if(n == indexMax)
230                     ++indexMax;
231
232             // Set PDU type - A (Acknoledgement)
233             tpdu.PDU_type = 'A';
234             printf("[R] Registration: File %s registered:\n",name);
235             printf("[R] Registration: Peer name: %s\n",user);
236             printf("[R] Registration: %s IP Address %s\n",user, inet_ntoa(p_addr.sin_addr));
237             printf("[R] Registration: %s port number %d\n",user, ntohs(p_addr.sin_port));
238             break;
239         }
240     }
241
242     // Send PDU Acknoledgement to client
243     sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
244
245 }else{
246     index = file_list[located].start;
247
248     // Checks for duplicated user names
249     while(index != NULL){
250         // If username is duplicated
251         if(strcmp(index->user, user) == 0){
252             duplicate = 1;
253             break;
254         }
255         else{
256             index = index->next; // Move to next node
257         }
258     }
259
260     // If duplicate is found
261     if(duplicate == 1){
262         free(temp); // Free memory allocated for temp
263
264         // Set PDU type - E (Error)
265         tpdu.PDU_type = 'E';
266         printf("(!) ERROR: Registration Duplicate Username Found%s\n", user);
267     }
268     else{ // User name is unique
269         index = file_list[located].start;
270         index->token = 0; // Set starting node token to 0
271
272         // Set tokens for all nodes to 0
273         while(index->next != NULL){
274             index = index->next;
275             index->token = 0;
276         }
277     }
278 }

```

```

279     index->next = temp;
280     temp->token=1;
281
282     // Set PDU type - A (Acknoledgement)
283     tpdu.PDU_type = 'A';
284
285     printf("[R] Registration: File %s registered:\n",name);
286     printf("[R] Registration: Peer name: %s\n",user);
287     printf("[R] Registration: %s IP Address %s\n",user, inet_ntoa(p_addr.sin_addr));
288     printf("[R] Registration: %s port number %d\n",user, ntohs(p_addr.sin_port));
289 }
290
291 // Send SPD response to the client
292 sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
293
294 }
295
296 /* SEARCH METHOD */
297 void search(int s, char *PDU_data, struct sockaddr_in *addr){
298
299     // Initialize Variables
300     int n,located=-1;
301     char name[FILENAMESIZE];
302     INPUT *start, *index;
303     PDU tpdu;
304
305     strcpy(name, PDU_data); // Copy content name from PDU data
306
307     // Find the file from the 'file_list' array
308     for(n=0; n<indexMax; n++){
309         // Check if file name matches
310         if((strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL) ){
311             located = n; // Record location
312             break;
313         }
314     }
315
316     // If File name could not be found
317     if(located == -1){
318         // Set PDU type - E (Error)
319         tpdu.PDU_type= 'E';
320         printf("[!] ERROR: Search was Unsuccessful\n");
321
322         //Send error PDU to client
323         sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
324         return;
325     }
326
327     else{ // Content is found
328         start = file_list[located].start;
329         index = start;
330
331         // find a peer with a token set to 1
332         while(index != NULL){
333             // if peer token = 1 is found
334             if(index->token == 1){
335                 printf("[+] Search: Located peer %s at port number %d\n", index->user,ntohs(index->addr.sin_port));
336                 index->token = 0; // Update tokens for the next search
337
338                 if(index->next == NULL){
339                     start->token=1;
340                 }
341                 else{
342                     index->next->token = 1;
343                 }
344
345                 // Set PDU types - S (Search)
346                 tpdu.PDU_type = 'S';
347
348                 memcpy(tpdu.PDU_data, &index->addr, sizeof(struct sockaddr)); // Copy peer address to the PDU data
349                 n = sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr)); // Send the PDU result to client
350             }
351             index = index->next;
352         }
353     }
354 }
355
356 /* DEREGRIGATION */
357 void deregistration(int s, char *PDU_data, struct sockaddr_in *addr){
358
359     // Initialize Variables
360     INPUT *index, *start, *prev;
361     int n, located = -1;
362     char name[FILENAMESIZE], user[FILENAMESIZE];
363     PDU tpdu;
364
365     //printf("Entering the deregistration routine\n");
366     strcpy(name, PDU_data); // Copy file name from recvPDU data
367     strcpy(user, &PDU_data[FILENAMESIZE]); // Copy user name from recvPDU data
368
369     // Find file from 'file_list' array
370     for(n=0; n<indexMax; n++){
371         // Check if file name matches

```

```

372     |     |     if((strcmp(name,file_list[n].name)==0) && (file_list[n].start!=NULL) ){
373     |     |     |     located= n;
374     |     |     |     break;
375     |     |
376     |
377 //printf("[T] Deregistration: at Location %d\n",located);
378 //printf("[T] Deregistration: the user %s\n", user);
380
381 // If the content for deregistration is not found
382 if(located == -1){
383
384     // Set PDU type = 'E'
385     tpdu.PDU_type = 'E';
386     // Send error PDU to client
387     sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
388     return;
389 }
390 else{
391     start = index = prev = file_list[located].start;
392
393     // Find input for the deregistration
394     while(index != NULL){
395         if(strcmp(index->user,user) == 0){
396             located = 1; // input is located when equal to 1
397             //printf("[T] Deregistration located the input\n");
398             printf("[T] Deregistration remove the input: %d\n", ntohs(index->addr.sin_port));
399
400             // If input that will be removed is the starting node
401             if(index == start){
402                 file_list[located].start = index->next;
403             }
404             else{
405                 // Remove the input from the linekd list
406                 prev->next = index->next;
407
408                 // If the removed node has the token
409                 if(index->token == 1){
410                     // Update token for the next deregistration
411                     if(index->next == NULL){
412                         file_list[located].start->token=1;
413                     }
414                     else{
415                         index->next->token = 1;
416                     }
417                 }
418             }
419
420             free(index); // free the memory allocation fro the removed input node
421             break;
422         }
423         prev = index;
424         index = index->next;
425     }
426
427     // Check in input was not located
428     if(located == -1){
429         // Set PDU type = 'E'
430         tpdu.PDU_type = 'E';
431     }
432     else{
433         // Check if starting node is null and located index to the last one
434         if(file_list[located].start == NULL && located == indexMax-1) indexMax--;
435         // Set PDU type = 'E'
436         tpdu.PDU_type='T';
437     }
438
439     // Send PDU response to the client
440     sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
441 }

```