

**Faculty of Engineering, Architecture and Science**

Department of Electrical and Computer Engineering

|               |                                |
|---------------|--------------------------------|
| Course Number | <b>COE 768</b>                 |
| Course Title  | <b>Computer Networks</b>       |
| Semester/Year | <b>Fall 2021</b>               |
| Instructor    | <b>Dr. Khalid Abdel Hafeez</b> |

|                                |          |
|--------------------------------|----------|
| <b>Lab/Tutorial Report NO.</b> | <b>6</b> |
|--------------------------------|----------|

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

|                 |             |
|-----------------|-------------|
| Section No.     | 3           |
| Group No.       |             |
| Submission Date | Nov 27 2021 |
| Due Date        | Nov 27 2021 |

| Name         | Student ID | Signature* |
|--------------|------------|------------|
| Toni Pano    | 500822828  | TP         |
| Jared Leaman | 500812326  | JL         |

(Note: remove the first 4 digits from your student ID)

*\*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, which can be found online at:*  
<http://www.ryerson.ca/senate/policies/pol60.pdf>

# Table of Contents

|                                       |           |
|---------------------------------------|-----------|
| <b>Introduction</b>                   | <b>3</b>  |
| <b>Description</b>                    | <b>3</b>  |
| Protocol Data Unit (PDU)              | 3         |
| Index Server                          | 4         |
| Content Registration                  | 5         |
| Content Deregistration                | 5         |
| Content List                          | 5         |
| Content Search                        | 5         |
| Flowchart of the Index Server Program | 6         |
| Peer                                  | 6         |
| General Functions                     | 6         |
| UDP Connection                        | 6         |
| TCP Connection                        | 7         |
| Listen to Multiple Sockets            | 7         |
| Local Content List                    | 8         |
| Client Functions                      | 8         |
| Index Server Content List             | 8         |
| Content Search and Download           | 8         |
| Content Server Functions              | 9         |
| Content Registration                  | 9         |
| Content Deregistration                | 9         |
| Content Transfer                      | 9         |
| Quit                                  | 10        |
| Flowchart of the Peer Program         | 11        |
| <b>Observations and Analysis</b>      | <b>12</b> |
| <b>Conclusion</b>                     | <b>17</b> |
| <b>References</b>                     | <b>17</b> |
| <b>Appendix A</b>                     | <b>17</b> |
| pdu.h                                 | 17        |
| index_serverv2.c                      | 17        |
| p2p_peer.c                            | 23        |

# 1. Introduction

The goal of this project is to curate all of the knowledge acquired from the previous labs and use it to create a server between multiple peers. This requires TCP and UDP connections to be created as well as a general data PDU with Data types and fields similar to those implemented in previous labs. To complete this task an index server and peer files were created to allow communication over a network. In general, the index server catalogues a list of connected peers and directs the peers to one another. The peers in turn are able to register and deregister content on the index server to act as a data host as well as search and download available content from other peers. The content flow is controlled through the first byte of the Data pdu sent by the peer to the index server and other peers. The data pdu also contains the subject file name, IP address and port number for the requested client. The backbone of this setup is based in Socket Programming.

At a basic level, Socket programming utilizes an IP address and a Port number to determine the source and destination of the data. For example a server with a unique IP address would be listening on a specific port which peers can connect to and communicate with the server. In this project the socket types that were used are TCP and UDP sockets. The TCP sockets connect peers to one another and the UDP socket is used for communication between the server and peer.

## 2. Description

### 2.1. Protocol Data Unit (PDU)

As defined in the protocol description of the project outline, the Protocol Data Unit defines the data and type of data that each data unit can contain. The type field of the PDU is 1 byte, and the data field of the PDU is at most 100 bytes long. The type field Each type of PDU stores a different arrangement of data in its data field.

Table 1. Arrangement of data for each type of PDU. Referenced from the COE768 project manual [1].

| PDU Description          | Fields    |                  |                                   |              |
|--------------------------|-----------|------------------|-----------------------------------|--------------|
|                          | Type (1B) | Data (max. 100B) |                                   |              |
| Content Registration     | R         | Peer Name (10B)  | Content Name (10B)                | Address (6B) |
| Content Download Request | D         | Peer Name (10B)  | Content Name (10B)                | Unused       |
| Content Search           | S         | Peer Name (10B)  | Content Name (10B) / Address (6B) | Unused       |

|                            |   |                          |                    |              |
|----------------------------|---|--------------------------|--------------------|--------------|
| Content Deregistration     | T | Peer Name (10B)          | Content Name (10B) | Address (6B) |
| Content Data               | C | File Data (100B Max)     |                    |              |
| List of Registered Content | O | Unused                   |                    |              |
| Acknowledgement            | A | Unused                   |                    |              |
| Error                      | E | Error Message (100B Max) |                    |              |

Note that the address field for a specified peer consists of the IP address and port of that peer. The address field is 6 bytes long. The first 4 bytes of the address field are the IP address in network byte order, and the last 2 bytes are the port in network byte order. The default order of bytes on a computer (host byte order) depends on the hardware each computer uses, and may be different between computers. This may cause issues when comparing different IP addresses and ports to each other. Instead we store them in network byte order to remain the same no matter what computer hardware is used [2].

Table 2. Arrangement of data within the address field. This applies to any PDU type which uses the address field.

| Address Field (6B)                  |                               |
|-------------------------------------|-------------------------------|
| IP Address (4B, network byte order) | Port (2B, network byte order) |

To implement the PDU, a struct of a char for the type field and an array of 100 chars for the data field is used. The PDU types are each defined as a macro variable using C's preprocessor directives. See Appendix A for the implementation of the PDU in a C file called "pdu.h".

The IP address and port are stored as an int type and short type respectively, and the PDU data field is an array of chars. To store the IP address and port in the data field, a specific index in the array is cast to an int for the IP address, and a short for the port.

## 2.2. Index Server

When the index server starts up it creates a UDP socket and binds itself to constantly listening on the bound port. When a peer connects to the server, it listens for incoming data and will either perform a content registration, content deregistration, search or content listing function, each of which are defined below. To store all of this information a linked list is created to allow seamless management of the content index listings on the server. See Appendix A for the implementation of the Index Server in a C file called "index\_serverv2.c".

### **2.2.1. Content Registration**

When the server receives a Data type 'R' from a client it performs a content registration. The server extracts the data PDU from the received packet and separates the data into the Peer Name, Content Name and Peer IP address and port. The server then searches the content registry to see if a peer with the same name and content has already been registered. If it has, the server sends an error to the client (Data type 'E' ) asking the client to change its name. If the content has not been registered before the server then adds the content to its linked list or "Index" of registered content and sends an Acknowledgement (Data type 'A') to the peer confirming that the data has successfully been registered.

### **2.2.2. Content Deregistration**

A content deregistration action occurs when a client either wants to deregister a single piece of content or wants to quit from the content cycle. When this happens the client sends a Data type 'T' to the server. The server then once again extracts the peer name, content name and peer ip from the data pdu sent. The server then searches through the index of registered content to find the content to be removed. If found, the server removes the content from the linked list and sends an acknowledgement to the client.

### **2.2.3. Content List**

The server performs a content list action when it receives a Data type 'O' from the client. When the server receives this request, the server iterates through the content list and sends each registered listing to the client. When the server has completely finished iterating through the list, it sends an acknowledgement to the client signalling that it is at the end of the list.

### **2.2.4. Content Search**

When the server receives a data type of type 'S' it performs the search function. The data pdu contains the content name to be searched. If the content is found, the peer that has been downloaded from the least is selected. The server then sends the address of that client to the requesting client using an 'S' type packet. If the content cannot be found it sends an Error type packet to the requesting client.

### 2.2.5. Flowchart of the Index Server Program

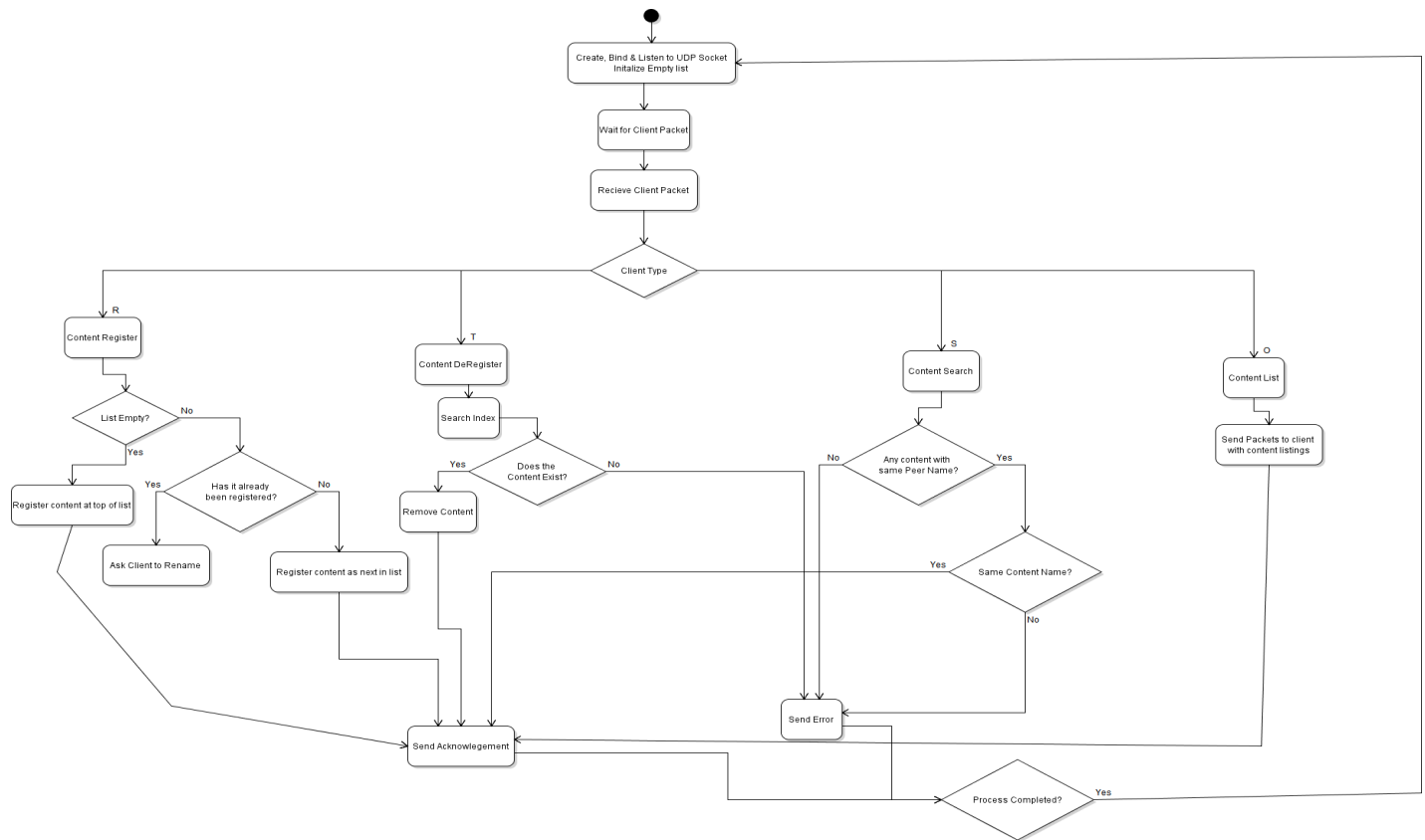


Figure 1. Flowchart of the Index Server

## 2.3. Peer

The peer implements two main functions. It acts as a content server to let other peers download any of its contents that are registered on the index server. It also acts as a client for a user to interact with the index server's services and to download content from other peer content servers. When interacting with the index server, it uses a UDP connection. When interacting with other peers, it uses a TCP connection. See Appendix A for the implementation of the peer in a C file called "**p2p\_peer.c**".

### 2.3.1. General Functions

Both the peer client and peer content server use the following functions.

#### 2.3.1.1. UDP Connection

When the peer starts, it creates a UDP connection to the index server for later use. The two command line arguments used when starting the peer represent the IP address and port

number of the index server it wants to connect to. It creates a UDP type socket, then tries to connect to the index server by using the command line arguments.

### 2.3.1.2. TCP Connection

When the peer starts, it creates a passive TCP socket for letting other peers connect to it and download content from it. The socket is created on any available port, and on IP address 127.0.0.1 (aka localhost), as shown in the demo video that is related to this project's manual [3]. The IP address and port number of that TCP socket must be obtained and included when registering content on the index server. This is done by using the `getsockname()` system call.

When other peers want to download a specific piece of content, they contact the index server for the IP address and port that they should connect to. They then form a TCP connection with the content server's passive TCP socket to download the desired content.

### 2.3.1.3. Listen to Multiple Sockets

The peer must respond to incoming TCP connections and user input from `stdin` simultaneously. The `read()` function can only listen for new input on `stdin`, an established TCP connection, or an established UDP connection. It will block until new input has been received. The `accept()` function will block until a new TCP connection has been made. Using either of these functions means the other function will not run until the code unblocks.

To avoid this problem, the `FD_SET()` functions can be used to form an interrupt for `stdin` and the TCP passive socket. The `afds` variable stores the interrupt flags that we want to respond to. The `memcpy()` function makes the `rfd`s variable store a temporary copy of those flags.

```
fd_set rfd, afds;

FD_ZERO(&afds);
FD_SET(listen_sd, &afds); //listening on a TCP socket
FD_SET(0, &afds); //listening on stdin

memcpy(&rfd, &afds, sizeof(rfd));
```

The `select()` function will then become non-zero when either input has something new to respond to. Once that happens, the `FD_ISSET()` function is used for each type of input to test whether it must respond to something. If it does, the flag corresponding to that input in the `rfd`s variable gets erased. After both input types have been tested, the `memcpy()` function resets the `rfd`s flag to what `afds` has stored, so that `select()` can respond to the `stdin` or TCP passive socket inputs again.

```
while(select(FD_SETSIZE, &rfd, NULL, NULL, NULL) > 0){
    if(FD_ISSET(0, &rfd)){ //stdin (0) has new input to read
        ...code to deal with user input...
    }
}
```

```

if(FD_ISSET(listen_sd, &rfdsets)){
    new_TCP_connection = accept(...);
    ...code to deal with new TCP connection...
}

//reset interrupt flags for stdin and listen_sd for select()?
memcpy(&rfdsets, &afds, sizeof(rfdsets));
}

```

#### 2.3.1.4. Local Content List

When the user types an 'L' character, the peer lists the content that is stored next to its "p2p\_peer.c" file. This is done by using the system("ls") system call. The "ls" command performs a list of content in the current directory on Linux systems.

### 2.3.2. Client Functions

The following functions are offered to the user of the peer for downloading content.

#### 2.3.2.1. Index Server Content List

When the user types an 'O' character, the peer sends an 'O' type PDU to the index server. The index server will send back an 'O' type PDU for each piece of content that has been registered on it. After the last 'O' type PDU has been sent from the index server to the peer, an 'A' type PDU will be sent to the peer to make the peer stop reading the UDP connection. Each 'O' type PDU will contain the peer name, content name, IP address, and port of a registered piece of content. Although this protocol does not follow the protocol for the "Content List" in the lab manual, it made debugging the program easier.

#### 2.3.2.2. Content Search and Download

When the user types an 'S' character, the peer sends an 'S' type PDU to the index server to search for any registered content with a specific name. The PDU's data field contains the name of the peer who sent the PDU and the name of the content which the peer wants to download. If the index server responds with an 'E' type PDU, it means no content with that name was registered on the index server. If the server responds with an 'S' type PDU, it will contain the IP address and port to download the content from. The peer will not allow the download of content from itself, if the IP address and port number from the 'S' type PDU match its own IP address and port.

Otherwise, the peer will form a new TCP connection using the address information. It first creates an empty file with the same name as the content it wants to download. Then it sends a 'D' type PDU containing its peer name and content name to ask the content server to download a specific piece of content. If the content server cannot find or read a file with the same content name in its directory, it responds with an 'E' type PDU to the peer, and the peer will delete the empty file. Otherwise the content server will respond with C type PDU's containing the file contents in the data field, and will close the TCP connection. The peer server



will continually read C type PDUs and store the contents in the data fields into the empty file until the TCP connection is closed.

Once all contents of the file have been downloaded successfully, the peer will attempt to register that file as its own content on the index server. The peer will send an 'R' type PDU to the index server that contains its peer name, content name of the file that it downloaded, IP address and port number.

### **2.3.3. Content Server Functions**

On startup the content server will accept incoming TCP requests and create a TCP connection between the peer server and peer client requesting the file. Once a connection has been established it provides the user with the opportunity to upload content to it and allow other peers to connect to it to download the hosted uploads.

#### **2.3.3.1. Content Registration**

When a user would like to register a piece of content to the index server, it is signified by a selection of type 'R' from the menu. The user is prompted to enter the content name they wish to register. The client then sends an 'R' type packet to the index server and waits for a response. If the client receives an 'A' type, the content is successfully registered, and the peer adds the content name to its local array of registered content names. If the client receives a 'E' type packet with the data field starting with a 'D', a duplicate piece of content by the peer has been found already registered and therefore the server does not register it. Otherwise, if the client receives an 'E' type packet, the index server already has a peer registered with its name in a separate client and the current peer is asked to change its name to avoid confusion.

#### **2.3.3.2. Content Deregistration**

When a user would like to deregister a piece of content, it is signified by a selection of 'T' in the menu. The user is then prompted to input which piece of content they would like to delist from the index server. When input, the client sends a 'T' type packet with the content name to the index server. The peer then waits for a packet from the index server. If the received packet contains an type 'A', the peer knows that the content has been deregistered successfully from the index server. If it receives a type 'E' packet, the peer understands that there has been an error when deregistering and therefore no such content with the content name sent is registered to the server by the peer.

#### **2.3.3.3. Content Transfer**

When a peer has searched for a piece of content and requests the hosted content from the content server, the content server receives a TCP connection request from the downloader and accepts. Once accepted, the content server listens for which file the client wishes to download. The server then takes the filename requested, searches for the file and if successful, opens the file. The server then reads the file into a buffer of 100 bytes and sends multiple 'C' type 100 byte packets to the downloader to initiate the file transfer. When the file transfer is complete, the server closes the TCP connection to the client, to tell the client that no more data

will be sent. Then the server then unregisters itself as the host of the content to make way for the downloader to become the host.

#### **2.3.3.4. Quit**

To quit out of the session the user selects 'Q' from the content menu. When a peer wishes to quit out of the content serving circuit it first must deregister all registered content listings from the index server. To do this, the peer sends a 'T' type packet to the index server for every name in its local array of registered content names. Each of these packets contains a single content listing to be deregistered from the index server. When the peer has no more names to send to the server, it stops the program. By this point, the index server should have deregistered all content listings of the peer.

## 2.3.4. Flowchart of the Peer Program

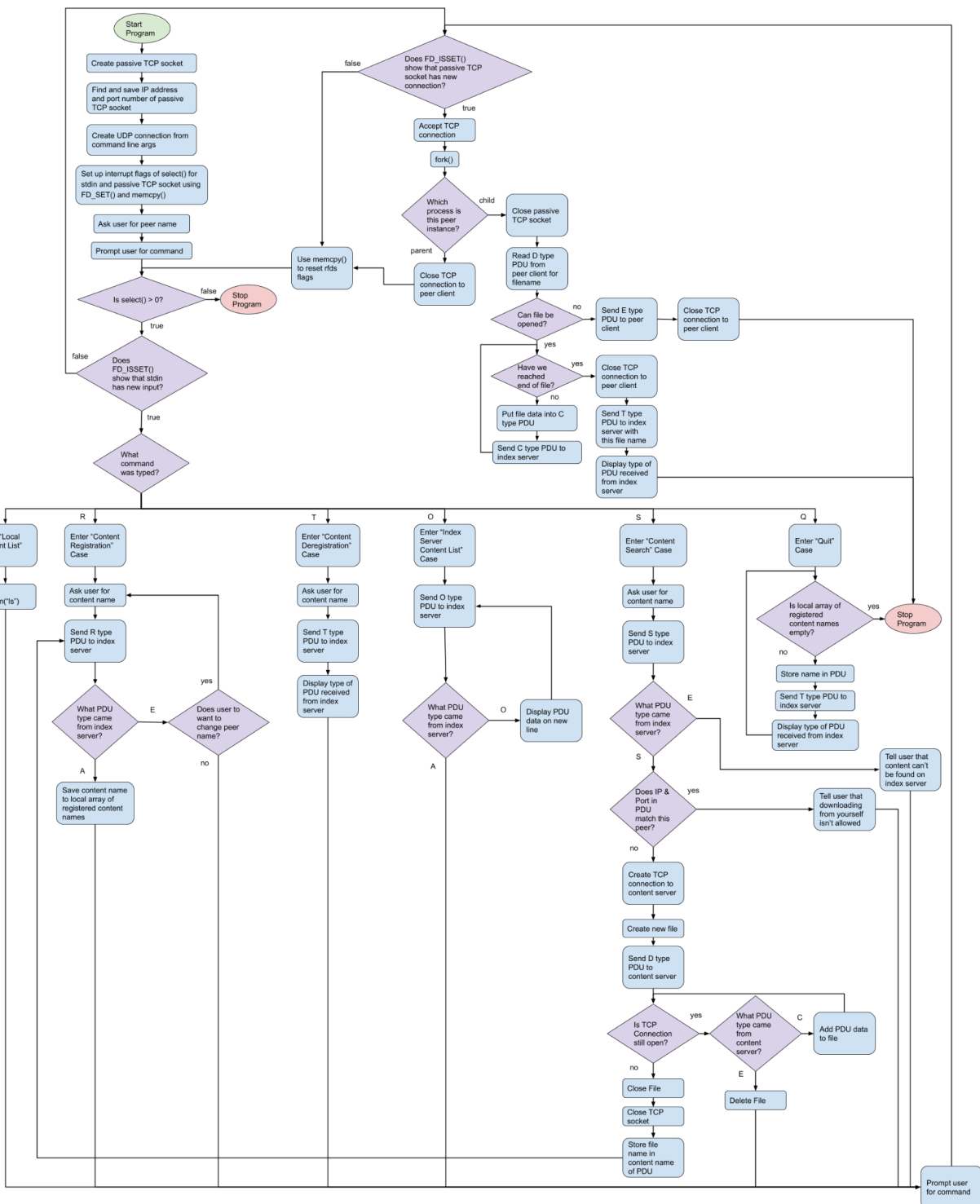


Figure 2. Flowchart of the Peer Client

### 3. Observations and Analysis

With the implementations discussed above, a series of systems connected to each other with TCP and UDP sockets in a client server form should be able to transfer data back and forth between each other.

In **Figure 3**. The startup screen of both the client and server are shown. The client start screen details the TCP and UDP connections that it has created and asks the user to enter its name as seen in **Figure 4**.

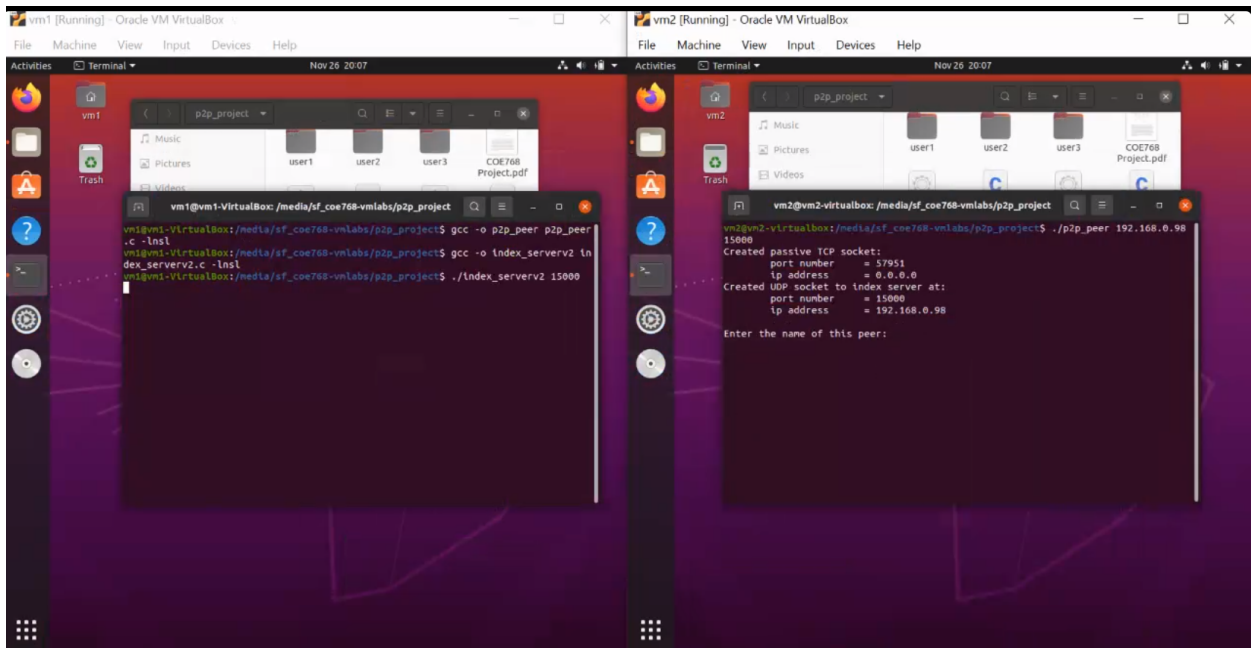


Figure 3. Start Up Screen of Server and Peer

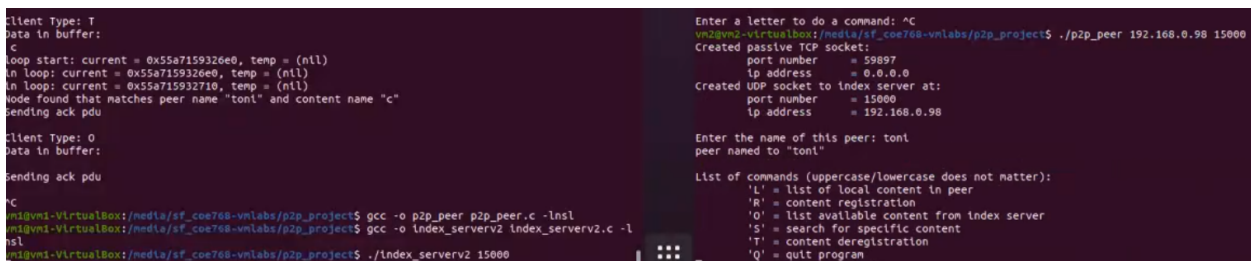


Figure 4. Example adding User toni

Once the user successfully registers its username, As shown in the above figure the server displays an acknowledgement that it has successfully registered the username and displays the menu of options to the user.

```

vm1@vm1-VirtualBox: /media/sf_coe768-vmlabs/p2p_project
List Has Items in It
Address (ip = 0, port = 57951) matches node
Current: 0x562a0395a6e0
Next: 0x562a0395a710
Address (ip = 0, port = 57951) matches node
Current: 0x562a0395a710
Next: 0x562a0395a740
Address (ip = 0, port = 57951) matches node
Current: 0x562a0395a740
Next: 0x562a0395a770
Address (ip = 0, port = 57951) matches node
Current: 0x562a0395a770
Next: (nil)
Inside Else
Current ID before set: (nil)
Before Address
Current ID after set: 0x562a0395a7a0
Client Type: A
Client Type: 0
Data in buffer:

vm2@vm2-virtualbox: /media/sf_coe768-vmlabs/p2p_project
Successfully registered content "b" from peer "toni"
Enter a letter to do a command: r
Enter the name of your content to register: c
Successfully registered content "c" from peer "toni"
Enter a letter to do a command: r
Enter the name of your content to register: d
Successfully registered content "d" from peer "toni"
Enter a letter to do a command: r
Enter the name of your content to register: e
Successfully registered content "e" from peer "toni"
Enter a letter to do a command: o
List of content on index server:
--- peer_name content_name ip port---
toni a 0 57951
toni b 0 57951
toni c 0 57951
toni d 0 57951
toni e 0 57951
Enter a letter to do a command:

```

**Figure 5. User toni adding Example content a,b,c,d,e to the server and listing**

The above **Figure 5** shows a peer, toni, registering example content a through e to the server and calling a list request by sending an 'O' type packet to the server. The server receives this 'O' packet and sends back multiple 'O' packets containing each item on the index server. The server then sends an acknowledgement, type 'A', packet to the peer to communicate that it has received the whole content list.

```

Current ID before set: (nil)
Before Address
Current ID after set: 0x562a0395a7a0
Client Type: A
Client Type: 0
Data in buffer:

Client Type: T
Data in buffer:
c
loop start: current = 0x562a0395a6e0, temp = (nil)
in loop: current = 0x562a0395a6e0, temp = (nil)
in loop: current = 0x562a0395a710, temp = (nil)
Client Type: 0
Data in buffer:

Client Type: 0
Data in buffer:

List of content on index server:
--- peer_name content_name ip port---
toni a 0 57951
toni b 0 57951
toni c 0 57951
toni d 0 57951
toni e 0 57951
Enter a letter to do a command: t
Enter the name of your content to deregister: c
Successfully deregistered content "c" from peer "toni"
Enter a letter to do a command: o
List of content on index server:
--- peer_name content_name ip port---
toni a 0 57951
toni b 0 57951
toni d 0 57951
toni e 0 57951
Enter a letter to do a command:

```

**Figure 6. User toni DeRegistering example content c from server and listing**

In **Figure 6** above, peer toni is shown to be deregistering content c through sending an 'O' type packet. The server acknowledges this, displays itself crawling through the index list to find the corresponding content to deregister. Once deregistered, the server sends an acknowledgement to the peer to indicate it was successful.

```

Enter the name of this peer: jared
peer named to "jared"

List of commands (uppercase/lowercase does not matter):
  'L' = list of local content in peer
  'R' = content registration
  'O' = list available content from index server
  'S' = search for specific content
  'T' = content deregistration
  'Q' = quit program

Enter a letter to do a command: o
List of content on index server:
--- peer_name content_name ip port---
    toni      a          0      36817
    toni      b          0      36817
    toni      c          0      36817

Enter a letter to do a command: r
Enter the name of your content to register: 1
Successfully registered content "1" from peer "jared"

Enter a letter to do a command: r
Enter the name of your content to register: 2
Successfully registered content "2" from peer "jared"

Enter a letter to do a command: r
Enter the name of your content to register: 3
Successfully registered content "3" from peer "jared"

Enter a letter to do a command: o
List of content on index server:
--- peer_name content_name ip port---
    toni      a          0      36817
    toni      b          0      36817
    toni      c          0      36817
    jared     1          0      35355
    jared     2          0      35355
    jared     3          0      35355

Enter a letter to do a command: █

```

**Figure 7. Peer jared created and registering content**

In the **Figure 7** above, the peer client is shown creating a second user jared. jared then registers example content 1 through 3 to the index server using an 'R' type packet. Then jared calls a content list using an 'O' type packet. When this is called the server successfully shows the different content indexed on the server and shows which user owns the content plus their respective TCP IP address and TCP port number.

```
terminal Nov 27 00:43
Index_serverV2.c
~/Desktop/Project/nov 27
Save
Index_serverV2.c
mncpy(head->add, s(client.data[20]),0); // copy
buffer to list
head->next = NULL;

coe768-1@coe7681-VirtualBox: ~/Desktop/Project/nov 27
Data in buffer:
test.txt
Head: (all)
BEFORE HEAD
BEFORE BREAK
Head Set to: 0x5609cd1d86e0

Client Type: S
Data in buffer:
test.txt

bu Client Type: R
Data in buffer:
test.txt

List Has Items in it
Current: 0x5609cd1d86e0
Next: (nil)
Inside Else
Current ID before set: (nil)
Before Address
Current ID after set: 0x5609cd1d8710
hes Client Type: A

ess match)

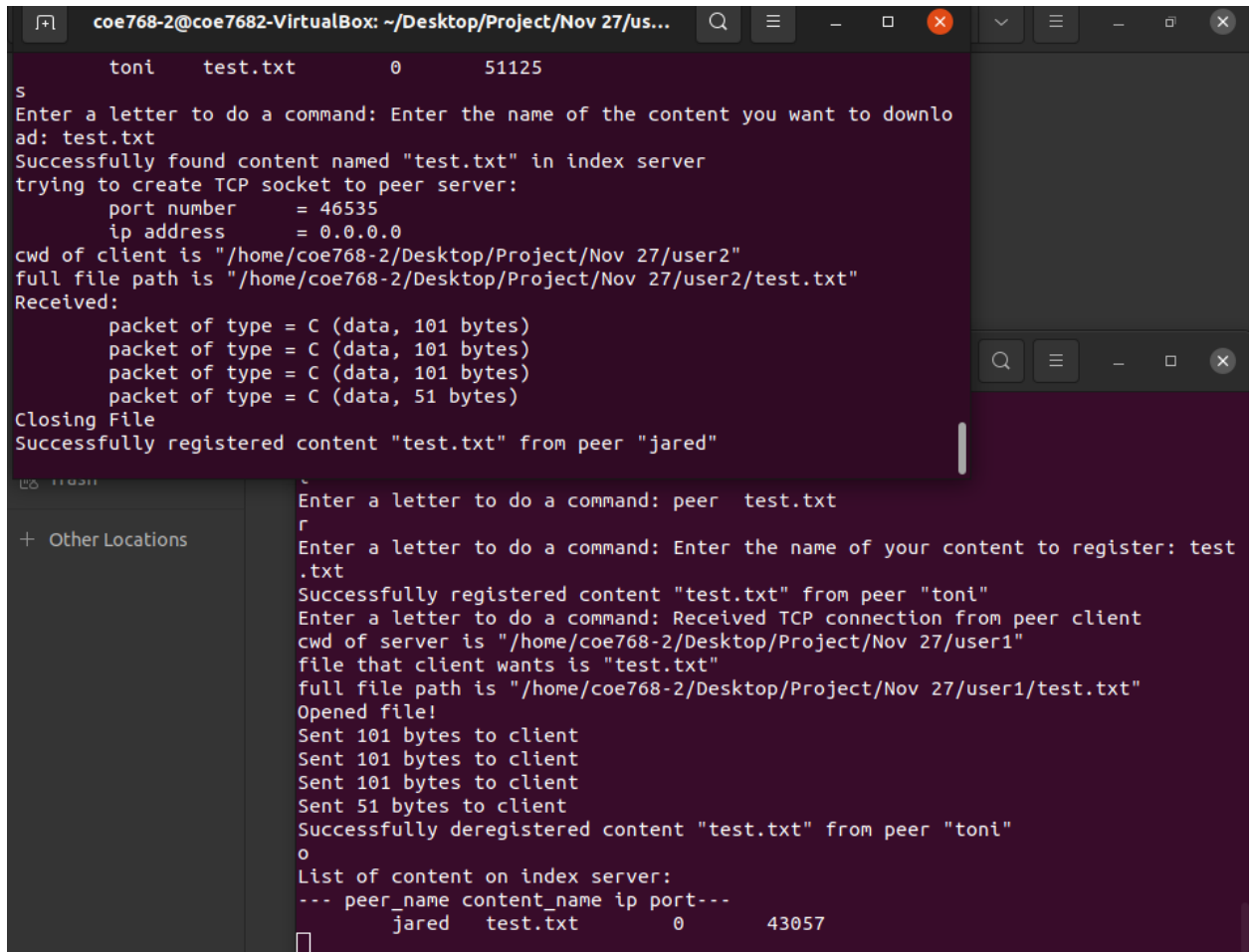
mp(checkcontent, current->cname) == 0)
{
    printf("Peer name \"%s\"
ent name \"%s\" matches node\n", current->pname, current->cname);
    dup = 1;
}

C ▾ Tab Width: 8 ▾ Ln1.Col
```

**Figure 8. Peer jared created and registering content**

Upon receiving the location, jared creates a TCP connection with toni and requests the file. When toni receives the request, toni transfers 'C' packets containing the file content in packets of 100 bytes. When the last packet is to be sent, toni sends a packet of type 'F' to signify to jared that it is ending the file transfer process. Once this packet is received, jared writes the file to disk and acknowledges a successful download.

Following this, in the below **Figure 9**, the process of deregistering the content from the previous content server and registering it to the client who downloaded it is shown. When user jared completes the download of the file, it registers itself as the new host of the file. When toni finishes sending the file, it deregisters itself as the host of the file to make sure that no duplicate files are being hosted on the index server.



```

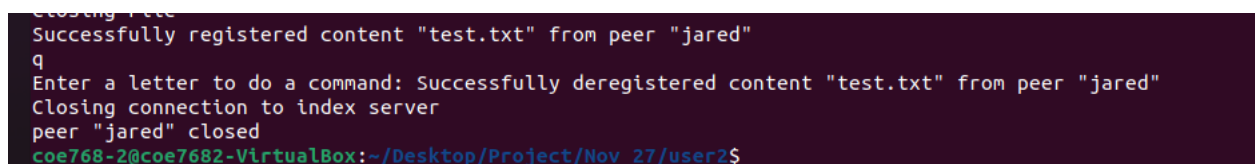
coe768-2@coe7682-VirtualBox: ~/Desktop/Project/Nov 27/us...
toni test.txt 0 51125
s
Enter a letter to do a command: Enter the name of the content you want to downlo
ad: test.txt
Successfully found content named "test.txt" in index server
trying to create TCP socket to peer server:
    port number      = 46535
    ip address       = 0.0.0.0
cwd of client is "/home/coe768-2/Desktop/Project/Nov 27/user2"
full file path is "/home/coe768-2/Desktop/Project/Nov 27/user2/test.txt"
Received:
    packet of type = C (data, 101 bytes)
    packet of type = C (data, 101 bytes)
    packet of type = C (data, 101 bytes)
    packet of type = C (data, 51 bytes)
Closing File
Successfully registered content "test.txt" from peer "jared"

Enter a letter to do a command: peer test.txt
r
Enter a letter to do a command: Enter the name of your content to register: test
.txt
Successfully registered content "test.txt" from peer "toni"
Enter a letter to do a command: Received TCP connection from peer client
cwd of server is "/home/coe768-2/Desktop/Project/Nov 27/user1"
file that client wants is "test.txt"
full file path is "/home/coe768-2/Desktop/Project/Nov 27/user1/test.txt"
Opened file!
Sent 101 bytes to client
Sent 101 bytes to client
Sent 101 bytes to client
Sent 51 bytes to client
Successfully deregistered content "test.txt" from peer "toni"
o
List of content on index server:
--- peer_name content_name ip port---
    jared test.txt 0 43057

q
Successfully deregistered content "test.txt" from peer "jared"
Closing connection to index server
peer "jared" closed
coe768-2@coe7682-VirtualBox:~/Desktop/Project/Nov 27/user2$

```

**Figure 9. Peer jared downloading and registering content**



```

Successfully registered content "test.txt" from peer "jared"
q
Enter a letter to do a command: Successfully deregistered content "test.txt" from peer "jared"
Closing connection to index server
peer "jared" closed
coe768-2@coe7682-VirtualBox:~/Desktop/Project/Nov 27/user2$

```

**Figure 10. Peer jared quitting**

Finally, when a peer wants to quit from the content server process, it is required to deregister all content hosted from the index server. **Figure 10** above shows peer jared initiating a 'Q' type quit request and successfully deregistering all content it hosts from the index server and exiting the client program.



## Conclusion

In conclusion, the goal of planning and creating a peer to peer server between multiple peers has been successfully accomplished. The knowledge from the previous labs helped accomplish the following tasks. Lab 3 helped set up a basic TCP connection between the Peer Content Server and Peer Client. Lab 4 aided in downloading files on a TCP connection for the Peer Content Server and Peer Client, and Lab 5 helped set up a basic UDP connection. Combining the knowledge gained and infrastructure used in the previous labs resulted in the successful creation of a peer-to-peer network, allowing for multiple peers to send and receive files from one another, directed by the content index server.

## References

[1] Dr. Khalid Abdel Hafeez, "P2P\_Project", *P2P Application [Online D2L]* Available <https://courses.ryerson.ca/d2l/le/content/521083/viewContent/3777674/View>

[2] IBM. Corp, "Network byte order and host byte order," *Network byte order and host Byte Order*. [Online]. Available: <https://www.ibm.com/docs/ja/zvm/7.2?topic=domains-network-byte-order-host-byte-order>. [Accessed: 28-Nov-2021].

[3] Dr. Ngok Wah (Bobby) Ma, "Peer to Peer Project Demo" *Peer to Peer Project Demo [Online Video D2L]* Available: <https://courses.ryerson.ca/d2l/le/content/521083/viewContent/3984974/View>

## Appendix A

### pdu.h

```
#define reg_pdu      'R' //Content Registration
#define dload_pdu   'D' //Download Request
#define search_pdu  'S' //Search for Content
#define dereg_pdu   'T' //Deregister Content
#define data_pdu     'C' //Content Data
#define list_pdu     'O' //Online List of Reg Content
#define ack_pdu      'A' //Acknowledgement
#define err_pdu      'E' //Error

struct pdu{
    char type;
    char data[100];
};
```

### index\_serverv2.c

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

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>
#include <time.h>
#include "pdu.h"

/*-----
 * main - Iterative UDP server for TIME service
 *-----
 */

typedef struct list {
    char pname[10], cname[10], add[10];
    struct list * next;
} list_t;

int
main(int argc, char *argv[])
{
    struct sockaddr_in fsin; /* the from address of a client */
    char buf[100]; /* "input" buffer; any size > 0 */
    char *pts;
    int sock; /* server socket */
    time_t now; /* current time */
    int alen; /* from-address length */
    struct sockaddr_in sin; /* an Internet endpoint address */
    int s, type; /* socket descriptor and socket type */
    int port=3000;
    int flag=0, dup = 0;
    struct pdu client;

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

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

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

    /* Bind the socket */
    if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        fprintf(stderr, "can't bind to %d port\n", port);
    listen(s, 5);
    alen = sizeof(fsin);

    list_t * head = NULL;

```

```

list_t * current;
list_t * previous;
//head = (list_t*) malloc(sizeof(list_t));
current = (list_t*)malloc(sizeof(list_t));
char checkpeer[10],checkcontent[10];
int checkip;
short checkport;

while (1) {

    if (recvfrom(s, &client, sizeof(client), 0,
                (struct sockaddr *)&fsin, &alen) < 0) //READ DIRECTLY INTO CLIENT
        fprintf(stderr, "recvfrom error\n");

    //client.type = buf[0]; // set the pdu type
    printf("Client Type: %c \n",client.type);
    //memmove(&(buf[0]),&(buf[1]),100);
    //client.data = buf;
    //strcpy(client.data,buf);
    printf("Data in buffer: \n %s \n",&(client.data[10]));

    if(client.type == reg_pdu) //Content Registration
    {
        /*
        Check if other peer with same name has registered the content aka
reregister

        if yes send error
        else register content into array
        */

        if(head == NULL) // First in List
        {
            printf("Head: %p\n", head);
            head = (list_t*)malloc(sizeof(list_t));
            printf("BEFORE HEAD\n");
            memcpy(head->pname, &(client.data[0]),10); // Copy peer
name from buffer to list
            memcpy(head->cname, &(client.data[10]),10); // Copy
Content name from buffer to list
            memcpy(head->add, &(client.data[20]),6); // Copy address
from buffer to list

            head->next = NULL;
            client.type = ack_pdu;
            (void) sendto(s, &client, sizeof(client), 0,
                (struct sockaddr *)&fsin, sizeof(fsin));
            printf("BEFORE BREAK\n");
            printf("Head Set to: %p\n", head);
            //break;
        }
        else{
            printf("List Has Items in it\n");
            current = head;

            memcpy(checkpeer, &(client.data[0]),10); // Copy peer
name from buffer to list
            memcpy(checkcontent, &(client.data[10]),10); // Copy peer
name from buffer to list

            checkip = *((int *) &(client.data[20]));
            checkport = *((short *) &(client.data[24]));

            while(current!=NULL)
            {

```

```

        if(checkip == *((int *) &(current->add)) &&
checkport == *((short *) &(current->add)[4])){ //if both peer name and content are same
        printf("Address (ip = %d, port = %d)
matches node\n", checkip, ntohs(checkport));

        //duplicate entry found (peer, content,
address match)
        if(strcmp(checkpeer,current->pname) == 0 &&
strcmp(checkcontent, current->cname) == 0)
        {
                printf("Peer name \"%s\" and content
name \"%s\" matches node\n", current->pname, current->cname);
                dup = 1;
                break;
        }
        }
        else{

                //addresses different but peer names match
means new peer using server. They must choose a different name
                if(strcmp(checkpeer, current->pname) == 0){
                        printf("Peer name \"%s\" matches
node\n", current->pname);

                        flag = 1;
                        break;
                }
        }

        printf("Current: %p\n Next: %p\n", current,
current->next);

        previous = current;
        current = current->next;
    }
    //current->next =(list_t*) malloc(sizeof(list_t));
    if(flag == 1){
        printf("Inside Flag  == 1\n");
        flag =0;
        client.type = err_pdu;
        printf("Client Type: %c \n",client.type);
        char err[] = "PEER NAME TAKEN";
        memcpy(&(client.data[0]),err,strlen(err)); //Copy
Content peer name into data
        (void) sendto(s, &client, sizeof(client), 0,
(struct sockaddr *)&fsin, sizeof(fsin));
    }
    else if(dup == 1){
        printf("Inside dup  == 1\n");
        dup = 0;
        client.type = err_pdu;
        printf("Client Type: %c \n",client.type);
        char err [] = "DUPLICATE ENTRY";
        memcpy(client.data, err, strlen(err));
        (void) sendto(s, &client, sizeof(client), 0,
(struct sockaddr *)&fsin, sizeof(fsin)); // Send ack
    }
    else{
        printf("Inside Else\n");
        printf("Current ID before set: %p\n", current);
        current =(list_t*) malloc(sizeof(list_t));

```

```

previous node                                previous->next = current;    //link this node to
Copy peer name from buffer to list           memcpy(current->pname, &(client.data[0]),10); //
Copy Content name from buffer to list        memcpy(current->cname, &(client.data[10]),10); //
Copy address from buffer to list             printf("Before Address\n");
                                              memcpy(current->add, &(client.data[20]),6); //
                                              current->next = NULL;
                                              printf("Current ID after set: %p\n", current);
                                              client.type = ack_pdu;
                                              printf("Client Type: %c \n",client.type);
                                              (void) sendto(s, &client, sizeof(client), 0,
(struct sockaddr *)&fsin, sizeof(fsin)); // Send ack
                                              }
                                              }
                                              }

if(client.type == search_pdu) //Search content listing for downloadable
content
{
    /*
    Peer contact index server and send S type PDU
    Extract peer name(10b) and content name(10b/end of data field)
    Check peer name in list and send S with Address if exists and E
if doesnt exist
    */
    //char checkpeer[10];
    current = head;
    int miss = 1;

    memcpy(checkcontent, &(client.data[10]),10);
    while(current!=NULL){
        if(strcmp(checkcontent,current->cname) == 0)
        {
            client.type = search_pdu;
            memcpy(&(client.data[20]),current->add,6); //Copy Content
server address into data
            (void) sendto(s, &client, sizeof(client), 0,
data with Search Type and address
                (struct sockaddr *)&fsin, sizeof(fsin)); // Send
                miss = 0;
                break;
            }
            current = current->next;
        }

        if(miss){
            client.type = err_pdu; //if failed send err pdu
            (void) sendto(s, &client, sizeof(client), 0, (struct
sockaddr *)&fsin, sizeof(fsin)); // Send Error
        }
    }

    if(client.type == dereg_pdu)
    {
        // GO through and remove content with associated name and content
        int miss = 1;                //flag if nothing in list matches
query

```

```

list_t * current = head;
list_t * temp = NULL; //is current
list_t * temp2 = NULL; //is current -> next
memcpy(checkpeer, &(client.data[0]),10); // Copy peer name from
buffer to list

memcpy(checkcontent, &(client.data[10]),10); // Copy peer name
from buffer to list

if(head != NULL){ //check list if not empty
    printf("loop start: current = %p, temp = %p\n", current,
temp);
    while(current->next != NULL)
    {
        printf("in loop: current = %p, temp = %p\n",
current, temp);
        if(strcmp(checkpeer,current->next->pname) == 0 &&
strcmp(checkcontent,current->next->cname) == 0)
        {
            temp = current->next; //node to
remove
            current->next = temp->next; //link node
after with node before, avoiding node to remove
            free(temp);
            miss = 0;
            printf("Node found that matches peer name
\"%s\" and content name \"%s\"\n", checkpeer, checkcontent);
            break;
        }
        current = current->next;
    }

    if(strcmp(checkpeer, head->pname) == 0 &&
strcmp(checkcontent, head->cname) == 0){
        printf("Head matches peer name \"%s\" and content
name \"%s\"\n", checkpeer, checkcontent);
        temp = head;
        head = head -> next;
        free(temp);
        miss = 0;
    }
}

if(miss){
    printf("Sending error pdu\n");
    client.type = err_pdu; //if failed send err pdu
    (void) sendto(s, &client,sizeof(client), 0, (struct
sockaddr *)&fsin, sizeof(fsin)); // Send Error
}
else{
    printf("Sending ack pdu\n");
    client.type = ack_pdu;
    sendto(s, &client, sizeof(client), 0, (struct sockaddr *)
&fsin, sizeof(fsin));
}

if(client.type == list_pdu)
{
    //Compile and send list of registered content

```

```

        current = head; // Set Pointer to top of list
        while(current!=NULL)
        {

            client.type = list_pdu;
            memcpy(&(client.data[0]),current->pname,10); //Copy Content peer
name into data
            memcpy(&(client.data[10]),current->cname,10); //Copy Content name
into data
            memcpy(&(client.data[20]),current->add,6); //Copy Content server
address into data

            (void) sendto(s, &client, sizeof(client), 0,
                (struct sockaddr *)&fsin, sizeof(fsin));

            current = current->next;
        }

        printf("Sending ack pdu\n");
        client.type = ack_pdu;
        sendto(s, &client, sizeof(client), 0, (struct sockaddr *)&fsin,
sizeof(fsin));
    }

    printf("\n");
}
}

```

## p2p\_peer.c

```

#include <stdio.h>           //printf()
#include <stdlib.h>          //exit()
#include <string.h>          //memset(), strncpy(), strcat()
#include <sys/socket.h>      //socket(), bind(), listen(), accept(), AF_INET, struct sockaddr,
struct sockaddr_in
#include <arpa/inet.h>       //htons(), htonl(), ntohs(), ntohl(), INADDR_ANY, plus everything from
<sys/socket.h>?
#include <unistd.h>          //fork(), close(), read(), write(), getcwd()
#include <fcntl.h>           //open(), close(), for files?
#include <ctype.h>           //toupper()
#include <sys/select.h>      //select()
#include <sys/time.h>

#include "pdu.h"            //defines data packet structure and packet types

#define BUFLen 256
#define CWDLEN 200
#define FILELEN 100

void getFilePath(char fileName[FILELEN], char * fullFilePath); //function declaration for
getting absolute file path of new file to create
int sendFile(int sd);       //function declaration for sending file data to a peer client

struct pdu pkt;             //stores packets sent to /received from server. Variable is reused
(overwritten) for each packet
char peer_name[10];         //stores name of this peer
char content_name[10];      //store names, etc, typed into user input
int index_server_sd;        //active UDP connectionless connection to index server

int main(int argc, char **argv){
    char ip_str[16];         //temporary string for displaying ip with periods
    char localhost_str[10] = "localhost"; //string which user types to use ip address of
local host

```

```

        int sockaddr_in_len = sizeof(struct sockaddr_in); //weird, but needed in this format
for accept()
        int index_server_ip, index_server_port;        //ip and port of index server from user
input
        int this_server_ip, this_server_port;        //ip and port of this peer as the content
server

        struct sockaddr_in server;    //struct reused for servers, has IP version, IP address,
port #
        struct sockaddr_in client;    //struct reused for clients, has IP version, IP address,
port #

        //file descriptors for sockets, aka "socket descriptor"
        int listen_sd;                //passive TCP socket for listening to new TCP requests from
peer clients
        int peer_server_sd;           //active TCP socket for downloading as client from another
peer as server
        int peer_client_sd;          //active TCP socket with connection to/from a server client

        fd_set rfd, afd;

        /*create passive TCP socket*/
        listen_sd = socket(AF_INET, SOCK_STREAM, 0);    //create software socket for TCP/IPv4
        if(listen_sd == -1){
            printf("Can't create a socket\n");
            exit(-1);
        }

        FD_ZERO(&afd);
        FD_SET(listen_sd, &afd); //listening on a TCP socket
        FD_SET(0, &afd); //listening on stdin
        memcpy(&rfd, &afd, sizeof(rfd));

        /*bind passive TCP socket to a port and IP address*/
        memset(&server, 0, sizeof(server));    //clear server struct contents
        server.sin_family = AF_INET;            //set protocol to IPv4
        server.sin_port = htons(0);            //0 means bind to any
available port
        server.sin_addr.s_addr = htonl(INADDR_ANY); //specify IP address of machine running this
program, htonl converts host bytes to network byte order in long
        if(bind(listen_sd, (struct sockaddr *)&server, sizeof(server)) == -1){ //bind socket to an
available physical port in hardware of the Network Internet Card
            printf("Can't bind port to socket\n");
            exit(-1);
        }

        /*use passive TCP socket for listening to TCP connection requests fro  other peers, queue
up to 5 requests at a time*/
        listen(listen_sd, 5);    //put socket in passive (server) mode to listen for TCP connection
requests (buffer size of 5 requests at a time)

        /*get port and ip address of the passive TCP connection*/
        getsockname(listen_sd, (struct sockaddr *)&server, &sockaddr_in_len);    //get
port and ip of passive socket
        this_server_port = server.sin_port;
        this_server_ip = server.sin_addr.s_addr;
        inet_ntop(AF_INET, &this_server_ip, ip_str, INET_ADDRSTRLEN); //convert ip address
number into string
        printf("Created passive TCP socket:\n\tport number\t= %d\n\tip address\t= %s\n",
ntohs(this_server_port), ip_str);

        /*get ip address and port number of index server from user input*/
        switch(argc){
            case 3:
                if(strcmp(argv[1], localhost_str) == 0){
                    index_server_ip = htonl(INADDR_ANY);
                    //printf("using localhost ip = %d\n", index_server_ip);
                }
                else{
                    inet_pton(AF_INET, argv[1], (void *)&index_server_ip);
//convert ip address in string to number, using IPv4 protocol format
                    if(index_server_ip <= 0){

```



```

        printf("Can't convert string into ip address\n");
        exit(-1);
    }

    index_server_port = htons(atoi(argv[2]));
    break;
default:
    printf("usage: %s [index server ip address] [index server port number]\n",
argv[0]);
    exit(-1);
}

/*create a UDP socket struct to connect to index server*/
index_server_sd = socket(AF_INET, SOCK_DGRAM, 0); //AF_INET = IPv4, SOCK_STREAM = UDP, 0
= use single default protocol if possible
if(index_server_sd == -1){
    printf("Can't create a socket\n");
    exit(-1);
}

/*connect to index server*/
memset(&server, 0, sizeof(server));
server.sin_family = AF_INET;
server.sin_port = index_server_port;
server.sin_addr.s_addr = index_server_ip;
if(connect(index_server_sd, (struct sockaddr *)&server, sizeof(server)) == -1){
    printf("Can't connect\n");
    exit(-1);
}
printf("Created UDP socket to index server at:\n\tport number\t= %s\n\tip address\t=
%s\n", argv[2], argv[1]);

/*main loop*/
char reg_contents[20][10]; //store list of content registered on server
int reg_content_ptr = 0;
char fullFilePath[CWDLLEN+FILELEN]; //full path to file (cwd/folder/fileName)
int fd; //file descriptor to created file
char input[10]; //store input from scanf()
char command; //store user's menu choice
int loop = 1;
int i, n;

/*get name of peer*/
printf("\nEnter the name of this peer: ");
scanf("%s", peer_name); //read user's command, note that scanf leaves whitespace in the
stdin buffer
printf("peer named to \"%s\"\n", peer_name);

printf("\nList of commands (uppercase/lowercase does not matter):\n\t'L' = list of
local content in peer\n\t'R' = content registration\n\t'O' = list available content from index
server\n\t'S' = search for specific content\n\t'T' = content deregistration\n\t'Q' = quit
program\n");

//get user input
printf("Enter a letter to do a command: ");

while(select(FD_SETSIZE, &rfd, NULL, NULL, NULL) > 0){
    if(FD_ISSET(0, &rfd)){ //stdin (0) has new input to read
        //printf("waiting for user input:");
        scanf("%s", input); //read user's command, note that scanf leaves
whitespace in the stdin buffer
        command = toupper(input[0]); //convert any input char to uppercase

        //check user input cases
        switch(command){
            case '?': //help user by listing menu commands and
re-reading user input
                printf("List of commands (uppercase/lowercase does not
matter):\n\t'L' = list of local content in peer\n\t'R' = content registration\n\t'O' = list

```

```

available content from index server\n\t'S' = search for specific content\n\t'T' = content
deregistration\n\t'Q' = quit program\n");
    break;

    case 'L': //list local content
        system("ls"); //call "ls" command from Linux terminal via
system call
        break;

    case reg_pdu: //'R' = register one specific content
        //send r type pdus until successful
        while(1){
            printf("Enter the name of your content to
register: ");

            scanf("%s", content_name);

            pkt.type = command;
            strncpy(pkt.data, peer_name, 10);
            strncpy(&(pkt.data[10]), content_name, 10);
            *((int *) &(pkt.data[20])) = this_server_ip;
            *((short *) &(pkt.data[24])) = this_server_port;

            write(index_server_sd, &pkt, sizeof(pkt)); //send
R type pdu to server
            read(index_server_sd, &pkt, sizeof(pkt)); //read
pdu that server sent in response

            if(pkt.type == err_pdu){//check if peer name is in
use
                if(pkt.data[0] == 'D'){
                    printf("Duplicate entry found. No
changes made to server list.\n");
                    break;
                }
                else{
                    printf("Error: %s\n", pkt.data);
                    printf("Name \"%s\" is in use by
another peer on the index server\nWould you like to use another name? (type 'y' for yes or any
char for no): ", peer_name);

                    scanf("%s", input);
                    command = toupper(input[0]);//

                    if(command == 'y' || command ==

                    printf("Enter a new name for
this peer: ");

                    scanf("%s", peer_name);//read
user's command, note that scanf leaves whitespace in the stdin buffer
                }
                else break;
            }
        }
        else if(pkt.type == ack_pdu){
            printf("Successfully registered content
\"%s\" from peer \"%s\".\n", content_name, peer_name);

            //save registered content name in array for
deregistering content during quitting
            strncpy(reg_contents[reg_content_ptr],
content_name, 10);

            if(reg_content_ptr < 20){
                reg_content_ptr++;
            }
            break;
        }
        else{
            printf("Received unknown pdu of type %c\n",
pkt.type);

            break;
        }
    }
}

```

```

        break;

    case list_pdu: // 'O' = list all content on index server
        memset(&pkt, 0, sizeof(pkt)); // clear packet contents
        pkt.type = command; // set packet type to 'O' for list type
        write(index_server_sd, &pkt, sizeof(pkt)); // send list
type packet to index server
        read(index_server_sd, &pkt, sizeof(pkt)); // receive list
of content from server

        // assumes content name is max. of 10 bytes long
        printf("List of content on index server:\n");
        printf("--- peer_name content_name ip port---\n");

        while(1){
            // printf("Received pdu of type %c\n", pkt.type);
            if(pkt.type == ack_pdu) break;

            inet_ntop(AF_INET, ((int *) &(pkt.data[20])),
ip_str, INET_ADDRSTRLEN);
            printf("\t%s\t%s\t%d\t%d\n", &(pkt.data[0]),
&(pkt.data[10]), *((int *) &(pkt.data[20])), ntohs(*((int *) &(pkt.data[24]))));
            read(index_server_sd, &pkt, sizeof(pkt));
        }
        break;

    case search_pdu: // 'S' = download one specific content
        printf("Enter the name of the content you want to
download: ");

        scanf("%s", content_name);

        pkt.type = command;
        strncpy(pkt.data, peer_name, 10);
        strncpy(&(pkt.data[10]), content_name, 10);

        write(index_server_sd, &pkt, sizeof(pkt));
        read(index_server_sd, &pkt, sizeof(pkt));

        if(pkt.type == err_pdu){
            printf("Error: %s\n", pkt.data);
            printf("No content named \"%s\" found in index
server\n", content_name);
            break;
        }
        else if(pkt.type == search_pdu){
            // inet_ntop(AF_INET, &(pkt.data[10]), ip_str,
INET_ADDRSTRLEN); // convert ip address number into string

            if(this_server_ip == *((int *) &(pkt.data[20])) &&
(short)this_server_port == *((short *) &(pkt.data[24]))){
                printf("This peer (\"%s\") should not try
to download \"%s\" from itself\n", peer_name, content_name);
                break;
            }

            printf("Successfully found content named \"%s\" in
index server\n", content_name);

            /* create a TCP socket struct */
            peer_server_sd = socket(AF_INET, SOCK_STREAM, 0);
// AF_INET = IPv4, SOCK_STREAM = TCP, 0 = use single default protocol if possible
            if(peer_server_sd == -1){
                printf("Can't create a socket\n");
                exit(-1);
            }

            // char string[7];
            // memcpy(string, &(pkt.data[10]), 7);

            /* connect to another peer offering its content as
a server */

```

```

server.sin_family = AF_INET;
server.sin_port = *((short*)&(pkt.data[24]));
//port as short from type S packet of chars
server.sin_addr.s_addr = *((int*)&(pkt.data[20]));
//ip address as int from type S packet of chars

inet_ntop(AF_INET, &(server.sin_addr.s_addr),
ip_str, INET_ADDRSTRLEN); //convert ip address number into string
printf("trying to create TCP socket to peer
server:\n\tport number\t= %d\n\tip address\t= %s\n", server.sin_port, ip_str);

if(connect(peer_server_sd, (struct sockaddr
*&server, sizeof(server)) == -1){
    printf("Can't connect\n");
    exit(-1);
}

//create local file to store downloaded content
getFilePath(&(pkt.data[10]), fullFilePath); //get
full absolute file path to location of this program file
FILE * ptr = fopen(fullFilePath, "w");
if(ptr == NULL)
printf("CANNOT CREATE FILE \n");

fd = 1;

//fd = open(fullFilePath,
O_CREAT|O_WRONLY|O_TRUNC|S_IRWXU); //O_CREAT = create file, O_WRONLY = write only, O_TRUNC =
overwrite any existing file, S_IRWXU = create file
if(fd == -1){
    printf("Can't create file!\n");
    exit(-1);
}

//download content from peer content server
pkt.type = dload_pdu;
strncpy(&(pkt.data[10]), content_name, 10);

write(peer_server_sd, &pkt, sizeof(pkt)); //send D
type packet to peer content server

printf("Received:\n");
while((n = read(peer_server_sd, &pkt,
sizeof(pkt))) > 0){
    printf("\tpacket of type = %c ", pkt.type);
    //exit if server sends error or unknown
    packet type
    if(pkt.type == data_pdu){
        printf("(data, %d bytes)\n", n);
        //write(fd, pkt.data, n-1);
        fwrite(pkt.data, 1, n-1, ptr);
    }
    else if(pkt.type == err_pdu){//prepare to
        delete created file if error packet
        //close(fd);
        fclose(ptr);
        fd = -1;
        printf("(error)\n");
    }
    else{
        printf("(unrecognized)\n");
    }
}

close(peer_server_sd); //close active TCP
connection to peer content server

//delete file if error packet, otherwise close
file normally
if(fd == -1){

```

```

        remove(fullFilePath); //delete file
    }
    else{
        printf("Closing File \n");
        fclose(ptr);
        //close(fd);
    }
}

file

peer's username already

content server for the recently downloaded content

content_name, 10);

this_server_ip;

this_server_port;

sizeof(pkt)); //send R type pdu to server

sizeof(pkt)); //read pdu that server sent in response

peer name is in use

pkt.data);

by another peer on the index server\n", peer_name);

this peer: ");

        if(pkt.type == err_pdu){ //check if
            printf("Error: %s\n",
                printf("Name \"%s\" is in use
                printf("Enter a new name for
                scanf("%s", peer_name);
            }
            else{
                break;
            }
        }

        if(pkt.type == ack_pdu){
            //save registered content name in
            if(reg_content_ptr < 20){
                reg_content_ptr++;
            }
            printf("Successfully registered
            content \"%s\" from peer \"%s\" \n", content_name, peer_name);
            break;
        }
        else{
            printf("Received unknown pdu of type
            break;
        }
    }
}
else{
    printf("Received unknown pdu of type %c\n",
        pkt.type);
}
break;

case dereg_pdu: // 'T' = deregister one specific content
    printf("Enter the name of your content to deregister: ");
    scanf("%s", content_name);

```

```

        pkt.type = command;
        strncpy(pkt.data, peer_name, 10);
        strncpy(&(pkt.data[10]), content_name, 10);

        write(index_server_sd, &pkt, sizeof(pkt)); //send T type
pdu to server
        read(index_server_sd, &pkt, sizeof(pkt)); //read pdu
that server sent in response

        if(pkt.type == ack_pdu){
            printf("Successfully deregistered content \"%s\"
from peer \"%s\"\\n", content_name, peer_name);
        }
        else if(pkt.type == err_pdu){
            printf("Error: %s\\n", pkt.data);
            printf("No content named \"%s\" registered with
peer \"%s\" in index server\\n", content_name, peer_name);
        }
        break;

        case 'Q': //deregister all remaining content and quit
            //send T type packet for every registered piee of content
from this peer

            strncpy(pkt.data, peer_name, 10);
            for(i = reg_content_ptr-1; i >= 0; i--){
                pkt.type = dereg_pdu;
                strncpy(&(pkt.data[10]), reg_contents[i], 10);
                //printf("current thing in list[%d]: %s\\n", i,
reg_contents[i]);

                write(index_server_sd, &pkt, sizeof(pkt)); //send
T type pdu to server
                read(index_server_sd, &pkt, sizeof(pkt)); //read
pdu that server sent in response

                if(pkt.type == ack_pdu){
                    printf("Successfully deregistered content
\"%s\" from peer \"%s\"\\n", reg_contents[i], peer_name);
                }
                else if(pkt.type == err_pdu){
                    printf("Error: %s\\n", pkt.data);
                    printf("No content named \"%s\" registered
with peer \"%s\" in index server\\n", reg_contents[i], peer_name);
                }
            }

            printf("Closing connection to index server\\n");
            close(index_server_sd); //close UDP connectionless
connection

            printf("peer \"%s\" closed\\n", peer_name);
            exit(0);

        case 'D':
            printf("Waiting for TCP connection from peer client\\n");
            peer_client_sd = accept(listen_sd, (struct sockaddr *)
&client, &sockaddr_in_len);

            //fork to let child process deal with TCP request, and
parent process to stay in main loop
            switch(fork()){
                case 0: //child process
                    close(listen_sd); //close passive TCP
socket
                    sendFile(peer_client_sd); //send file
with content to peer client
                    close(peer_client_sd); //close TCP socket
to peer client after download is done
                    exit(0);
                default: //parent process
                    close(peer_client_sd); //close active TCP
socket to peer client asking for download

```

```

                                break;
                                case -1: //fork returns error code
                                    printf("fork: error\n");
                                    exit(-1);
                                }
                                break;

                                default:
                                    printf("command '%c' unrecognized. Type '?' for help with
commands.\n", command);
                                    break;
                                }

                                //get user input
                                printf("Enter a letter to do a command: ");

                                //register stdin for select() again?
                                //FD_ZERO(&afds);
                                //FD_SET(0, &afds); //listening on stdin
                                //memcpy(&rfd, &afds, sizeof(rfd));
                                }

                                if(FD_ISSET(listen_sd, &rfd)){ //listen_sd has at least 1 new connection
request
                                    printf("Received TCP connection from peer client\n");
                                    peer_client_sd = accept(listen_sd, (struct sockaddr *) &client,
&sockaddr_in_len);

                                    //fork to let child process deal with TCP request, and parent process to
stay in main loop
                                    switch(fork()){
                                        case 0://child process
                                            close(listen_sd); //close passive TCP socket
                                            sendFile(peer_client_sd); //send file with content to
peer client
                                            close(peer_client_sd); //close TCP socket to peer client
after download is done
                                            exit(0);
                                        default: //parent process
                                            close(peer_client_sd); //close active TCP socket to peer
client asking for download
                                            //FD_ZERO(&afds);
                                            //FD_SET(listen_sd, &afds); //listening on a TCP socket
                                            //memcpy(&rfd, &afds, sizeof(rfd));
                                            break;
                                        case -1: //fork returns error code
                                            printf("fork: error\n");
                                            exit(-1);
                                        }
                                    }

                                    //reset interrupt flags for stdin and listen_sd for select()?
                                    memcpy(&rfd, &afds, sizeof(rfd));
                                }
                            }

//function for getting absolute file path of new file to create
void getFilePath(char fileName[FILELEN], char * fullFilePath){
    char cwd[CWDLEN]; //current working directory of client program
    char slash[2] = "/";

    /*get current working directory, store in full file path*/
    getcwd(cwd, sizeof(cwd));
    strncpy(fullFilePath, cwd, CWDLEN);
    printf("cwd of client is \"%s\"\n", cwd);

    /*add slash after cwd*/
    strcat(fullFilePath, slash);

    /*add file name to end of cwd, store in full file path*/
    strcat(fullFilePath, fileName);

```

```

    printf("full file path is \"%s\"\n", fullFilePath);
}

//reads file contents to peer client
int sendFile(int sd){
    int n; //number of bytes that have been read in from socket sd
    struct pdu pkt; //data packet struct
    int fd; //file descriptor
    char cwd[CWDLEN]; //current working directory of server program
    char slash [2] = "/";
    char fullFilePath[CWDLEN+sizeof(pkt)]; //full path to file (cwd+fileName)
    char fileName[10];

    /*get current working directory, store in full file path*/
    getcwd(cwd, sizeof(cwd));
    strncpy(fullFilePath, cwd, CWDLEN);
    printf("cwd of server is \"%s\"\n", cwd);

    /*add slash after cwd*/
    strcat(fullFilePath, slash);

    /*get file name from client*/
    read(sd, &pkt, sizeof(pkt));
    printf("file that client wants is \"%s\"\n", &(pkt.data[10]));
    strncpy(fileName, &(pkt.data[10]), 10);

    /*add file name to end of cwd in full file path*/
    strcat(fullFilePath, &(pkt.data[10]));
    printf("full file path is \"%s\"\n", fullFilePath);

    /*search for file*/
    fd = open(fullFilePath, O_RDONLY);
    if(fd == -1){
        printf("Can't open file!\n");
        pkt.type = err_pdu;
        write(sd, &pkt, sizeof(pkt));
        close(fd);
        return -1;
    }
    else{
        printf("Opened file!\n");
        pkt.type = data_pdu;
        while((n = read(fd, pkt.data, sizeof(pkt.data))) > 0){
            write(sd, &pkt, n+1);
            printf("Sent %d bytes to client\n", n+1);
        }
        close(fd);

        pkt.type = dereg_pdu;
        strncpy(pkt.data, peer_name, 10);
        strncpy(&(pkt.data[10]), fileName, 10);

        write(index_server_sd, &pkt, sizeof(pkt)); //send T type pdu to server
        read(index_server_sd, &pkt, sizeof(pkt)); //read pdu that server sent in
response

        if(pkt.type == ack_pdu){
            printf("Successfully deregistered content \"%s\" from peer \"%s\"\n",
fileName, peer_name);
        }
        else if(pkt.type == err_pdu){
            printf("Error: %s\n", pkt.data);
            printf("No content named \"%s\" registered with peer \"%s\" in index
server\n", fileName, peer_name);
        }

        return 0;
    }
}
}

```