




## Faculty of Engineering and Architectural Science

### Department of Electrical and Computer Engineering

<b>Course Number</b>	COE 768
<b>Course Title</b>	Computer Networks
<b>Semester/Year</b>	F2020
<b>Project Title</b>	P2P Application
<b>Instructor Name</b>	Bobby Ma
<b>Section No</b>	05

<b>Submission Date</b>	12/05/2020
<b>Due Date</b>	12/04/2020

<b>Name</b>	<b>Student ID</b>	<b>Signature*</b>
Dimple Gamnani	500726015	
Vatsal Shreekant	500771363	

*\*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:*

[www.ryerson.ca/senate/current/pol60.pdf](http://www.ryerson.ca/senate/current/pol60.pdf)

## Contents

Introduction.....	3
What the Project is About .....	3
Background Information on Socket Programming .....	3
Description of the Client and Server Programs.....	3
Basic Approach to Implement the Protocol .....	3
Detailed Description of the Index Server Program.....	4
Detailed Description of the Client Program.....	5
Observations and Analysis.....	7
Test 1: Inputting a username & content registration.....	7
Test 2: Verify the existence of the registered file .....	7
Test 3: Verify the address of the file.....	8
Test 4: Attempt to Download Files .....	8
Test 5: Third user creation .....	9
Test 6: De-Register Files with Original Username .....	9
Conclusions.....	10
References.....	10
APPENDICES .....	11
P2Pserver Source Code.....	11
P2Ppeer Source Code.....	21

# Introduction

## What the Project is About

The objective of this project is to develop and implement the code for a basic peer-to-peer (P2P) file sharing application. The basic idea of a P2P file-sharing network is that many computers come together and pool their resources to form a content distribution system. The computers are called peers because each one can alternatively act as a client to another peer, fetching its content, and as a server, providing content to other peers [1]. P2P technology was used by popular services such as Napster and LimeWire. The most popular protocol for P2P sharing is BitTorrent [2].

For this project, peers can exchange content among themselves through the support of the index server. A peer that has a piece of content (a movie, a song or a text file) available for download by other peers is called the content server of that content.

## Background Information on Socket Programming

The concept of the socket is vital to network programming. Processes that want to communicate with one another need an interface. Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket (node) listens on a port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

## Description of the Client and Server Programs

### Basic Approach to Implement the Protocol

To initiate the protocol, the index server should be initiated, if a peer client wants to communicate with it. Next, a peer with a file that other peers want to register with the index server may proceed to download. The file is available for download once it has been registered. The subsequent peers may gain access to the file by requesting the server for the list of registered files. Next, the client user is asked for the file name to be downloaded. If the file is present in the index server, the server then arranges for the file transfer to the client, given that the file is available. The client peer then registers its request of the file with the index server and therefore imitates a content server. Any clients that may want to gain access to the file post this process, they would then be pointed to the latest peer to re-register the file. If a content server wants to de-register a file, it would do so by querying the index server.

Protocol data units (PDUs) are packets of information, that are being transferred back and forth between hosts. As per the specifications in the lab manual, each PDU is comprised of 100-byte portion, where data is stored. The 100-byte portion may in turn have subsequent sub-parts for multiple bytes of information. For the purposes of implementation of this project, the PDU will be represented by a typedef struct data structure.

Transmitting and receiving PDUs, socket objects must be bound to the appropriate host address. TCP protocol ensures a safe, reliable and secure transmission. A connection between the hosts from their sockets must be made. UDP protocol leads to a more insecure and unreliable transmission, and thus a connection is not made between the hosts. Typedef structures cannot be sent through a socket because a socket only accepts data in a binary format. Hence, the PDUs must be broadcasted into byte form for transmission, then de-broadcasted back into object form. The PDUs are sent by invoking '&rpdu'.

When a PDU is received, the receiving host must make decisions based largely on they label of the PDU and/or the presence or not of data. A table of instructions based on the PDU label is given below.

**Table 1: Client Directions based on PDU Label**

<b>PDU Type</b>	<b>Function</b>	<b>Direction</b>
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

Once there is no more data, or in some cases when a timeout exception is triggered, the reading/writing operation stops. If no data is received at all when it is expected, an exception is triggered, and an error message is returned to the sender.

Various custom functions calls were called to implement the protocol. The table[MAXCON] structs was made to keep track of the registered content. The function memset() was used to fill a block of memory with a certain values and to setup a UDP connection with the server. The socket() function was used as a file descriptor and direct a socket for the index. The implementation of select structure and table structure was done by: FD\_ZERO() to begin the file descriptor; FD\_SET() was used for listening on the index server socket and listening on the read descriptor.

### Detailed Description of the Index Server Program

The index server program is established for maintaining the list of online registration content, maintaining the registering and de-registering of files, and redirecting search requests to the server. The server code begins with preliminary definitions such as defining the PDU and the files list as types of typedef structs, setting the port number and client ID, and creating a UDP socket. An important task of the server in this project is its ability to direct various client requests simultaneously.

The index server's operation initiates by binding the socket object to the client's IP address and port number. During an error, the exception is initiated. The server then establishes that it is waiting for a connection and starts listening for peers. Upon a connection request acceptance and a connection made, the index server provides the IP address and port number of the client that it is communicating with.

Upon establishing a stable network connection, the packet is then received in a binary format and converted back into a PDU typedef struct. An acknowledgement message is printed upon receiving the specific PDU from a peer. The label of the PDU is checked for PDU type. After binding the socket, the type could be "Content Registration Request- R", "Search Content- S", "List current Content- O" or "De-registration- T".

If the type is "R", then the peer name, filename, and peer address are extracted and stored in variables. A "registration()" function is called with parameters 's, rpdu.data, & fsin' (to assume that the file to be registered does not already exist in the registry). The indexes in the file registration list process is matched against the established client/file name. If there is a match(list[n].head!=NULL), then the "find" variable is set to "n" and a PDU with an error message is sent back to the client telling them to pick

another username to register that file. If the file does not exist, then the client name, filename, and peer address are added to the server file registration list and an acknowledgement PDU is sent back to the client.

Upon entering “S” command in the terminal, the file name is stored in a variable. A “search\_content()” function is called with parameters ‘s, rpdu.data, & fsin’. A for-loop is run to search the file name. If the result is a negative integer then a message is print “Search Unsuccessful” and tpdu.type is set to 'E'. If there is a match, then a while loop is run and the tpdu.type is set to 'S' and the address of the matched file name is stored. An S-type acknowledgement PDU is sent to the client.

Upon entering “O” command in the terminal, the server first checks to see if there are any entries in the server file registration list. A message stating that such a PDU is being sent to the peer is printed. If there are entries in the list, then the client/file name and client address of the entry are sent to the client in an O-type PDU.

Upon entering “T” command in the terminal, the client/file name, and peer address are stored in variables. A “deregistration()” function is called with parameters ‘s, rpdu.data, & fsin’. The indexes in the file registration process list is matched for the established client/file name. upon receiving a match, the matching entry is deleted from the server file list, and acknowledgement PDU is sent to the client. Finally, if there are no more entries in the server file registration list, then an error PDU is sent to the client stating that the file does not exist.

## Detailed Description of the Client Program

The client sends connection requests to the associated server. The main responsibility of the client program are as follows: Sending requests to the server to either register/de-register; Requesting the list of online registered files; Choosing the requested file for download; Gathering the list of local registered content files; Post the deregistering process, to quit the current session. The client has also been afforded the ability to act as a content server.

Like the server code, the P2P client code also starts with basic definitions and declarations. The peer client code begins with elementary definitions such as defining the PDU and the files list as types of typedef structs, setting the port number and server ID, and creating a UDP socket

The code for the client side is comprised of the following function definitions: search\_content(), deregistration(), registration(), client\_download(), server\_download(). The process begins with a prompt for the user to enter a username. The client then starts both client and server function and calls the commands.

The user is asked to choose a function from the list: “O” (On-Line Registered Content); “L” (list local files); “R” (register a file); “T” (de-register a file); or “Q” (quit the program). The search\_content() function takes the following parameters: int s\_sock, char \*name, PDU \*rpdu. It sets the tpdu.type to 'S' and copies it onto the memory using the memcpy() function. Apart from searching the content and reading and writing error reports, it will print “Search\_Content: Cannot find content” for rpdu->type == 'E', “Search: Find content” for rpdu->type == 'S', and “Search\_Content: Protocol Error” if neither of these cases are true.

Upon entering “O” command in the terminal, the “online\_list()” function is called with the parameter s\_sock(socket) and an O-type PDU is sent to the index server. The responding PDU from the index server is downloaded signalling download completion. If the received PDU is Type-O, then its

package is received. If there are entries in the list, then the peer name and file name of each entry is displayed, and the user is prompted for which file to download.

If the receiving PDU is an “S” type, then the client address where the file is stored is taken from the package and the `server_download()` function is called, with the client address, filename request, and file index list as arguments.

Upon entering “L” command in the terminal, then the `local_list()` function is called with the folder containing the files as an argument, and the files are printed out with the message “Local Content List”. This function prints the file upon searching and matching the table of entries.

Upon entering “R” command in the terminal, the user is asked for the name of the file to register by printing a prompt message “Enter the file name to be registered” and then the `registration()` function is called with the username, filename index request, and client port number as parameters.

Upon entering “T” command in the terminal, the user is asked for the name of the file to de-registered. If the file exists, the `deregistration()` function is called with the same function parameters as the `registration()` function.

A P2P client can also imitate a server if it has registered a file with the index server. Basic declarations include establishing the TCP socket for connection between the clients and the sockets of clients requesting a communication.

The content server will listen for available sockets which are ready through the `FD_SET()` function. The content server will check the socket list for sockets to establish a connection. When a new connection is made with a peer, the peer gets added to the socket list.

Upon entering “D” command in the terminal, the server will match the address for the file. If the file is there, it will be sent to the client that asked for it in a PDU.

An error with PDU “E” will be generated if the server cannot locate the file. The connection would then be closed.

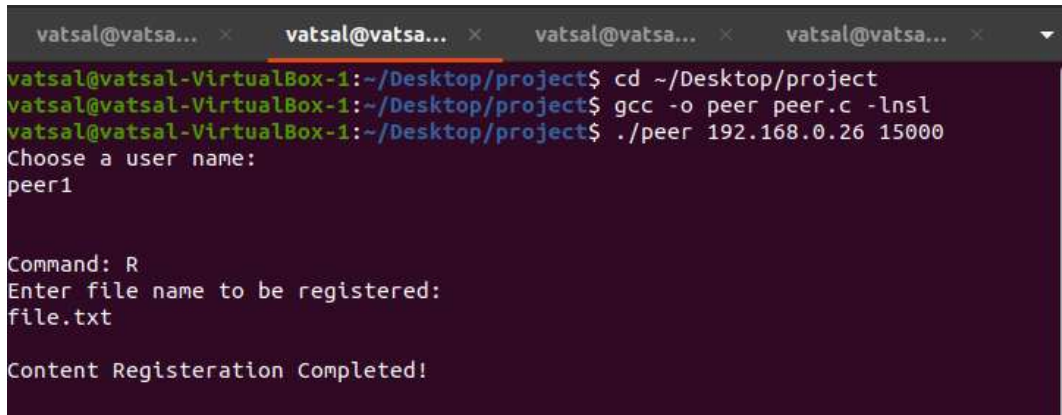
The P2P client implements three functions: file registration- `registration()`, download- `server_download()`, and de-registration- `deregistration()`. The `registration()` function takes two arguments: filename and client port. When the function is called, an R-type PDU is established and forwarded to the server. The package from the index server is then downloaded. The PDU type is then verified. Upon reading an “A” type, the filename is attached to the file and the package contents are displayed. Upon reading an “E” type, the error message is printed out. The new “R”-type PDU is prepared and sent.

## Observations and Analysis

The following are tests of basic functionality.

### Test 1: Inputting a username & content registration

This is a reasonable preliminary test to show a file existing and a file being registered. The R command is expected to succeed.

A terminal window with four tabs, all labeled 'vatsal@vatsa...'. The active tab shows the following commands and output:

```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer1

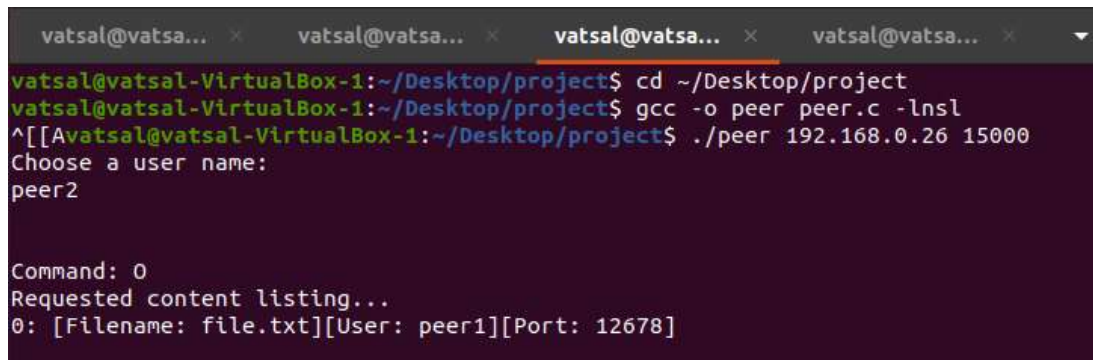
Command: R
Enter file name to be registered:
file.txt

Content Registration Completed!
```

*Figure 1: R-Type Command in Terminal by Peer1*

### Test 2: Verify the existence of the registered file

With entries in the registry, a call to view the registry or download files should be successful. The test is expected to succeed.

A terminal window with four tabs, all labeled 'vatsal@vatsa...'. The active tab shows the following commands and output:

```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
^[[Avatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer2

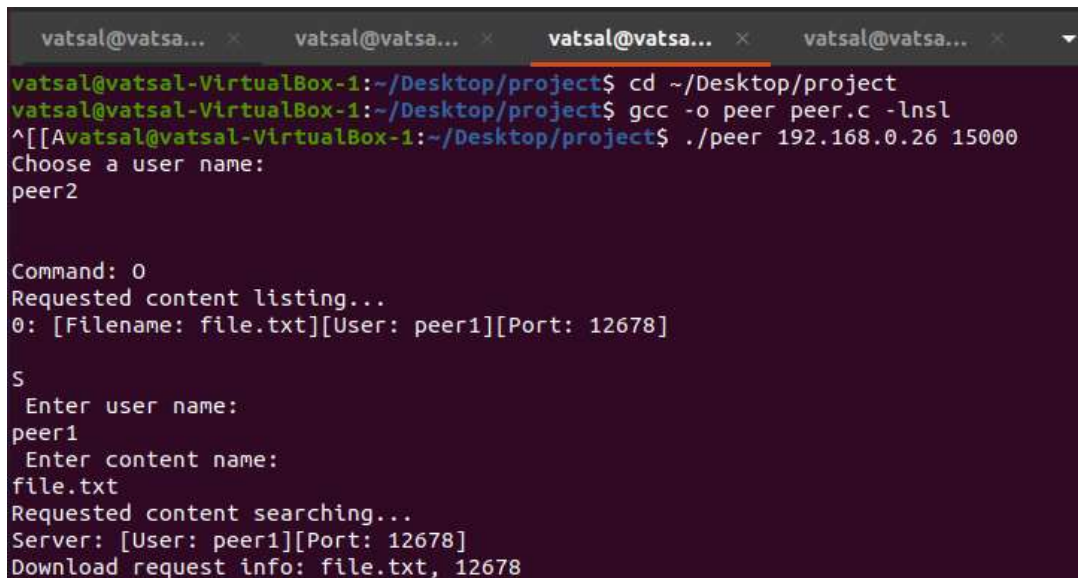
Command: O
Requested content listing...
0: [Filename: file.txt][User: peer1][Port: 12678]
```

*Figure 2: O-Type Command in Terminal by Peer2*



### Test 3: Verify the address of the file

The address of the file must first be found by creating another peer. An S is entered which requests to search for a specific file by inputting the username as well as the content name



```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
^[[Avatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer2

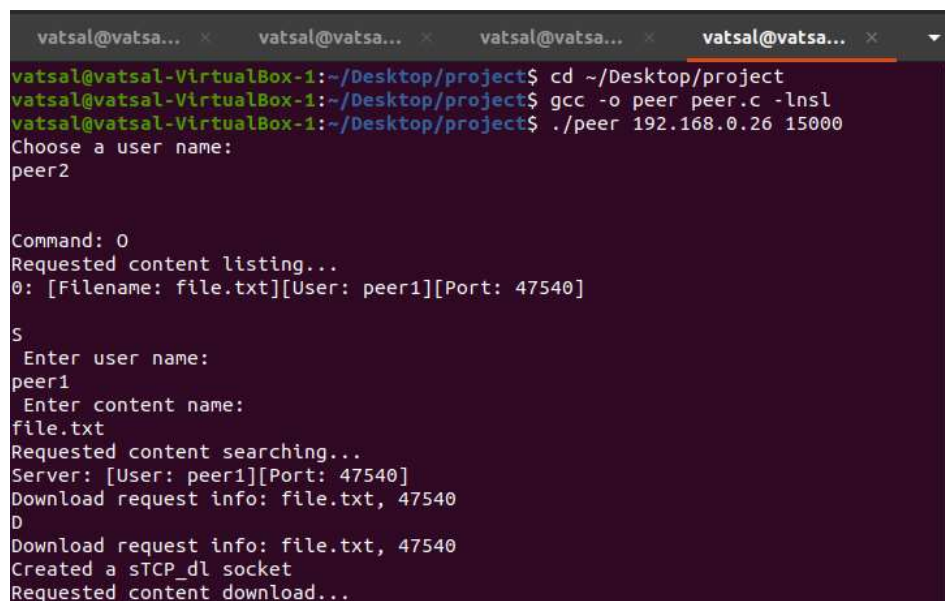
Command: 0
Requested content listing...
0: [Filename: file.txt][User: peer1][Port: 12678]

S
  Enter user name:
peer1
  Enter content name:
file.txt
Requested content searching...
Server: [User: peer1][Port: 12678]
Download request info: file.txt, 12678
```

*Figure 3: S-Type Command in Terminal by Peer2*

### Test 4: Attempt to Download Files

The code was designed to block peers with the same username from downloading the same file but allow peers with a different username to download and register the same file. The test should be successful.



```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer2

Command: 0
Requested content listing...
0: [Filename: file.txt][User: peer1][Port: 47540]

S
  Enter user name:
peer1
  Enter content name:
file.txt
Requested content searching...
Server: [User: peer1][Port: 47540]
Download request info: file.txt, 47540

D
Download request info: file.txt, 47540
Created a sTCP_dl socket
Requested content download...
```

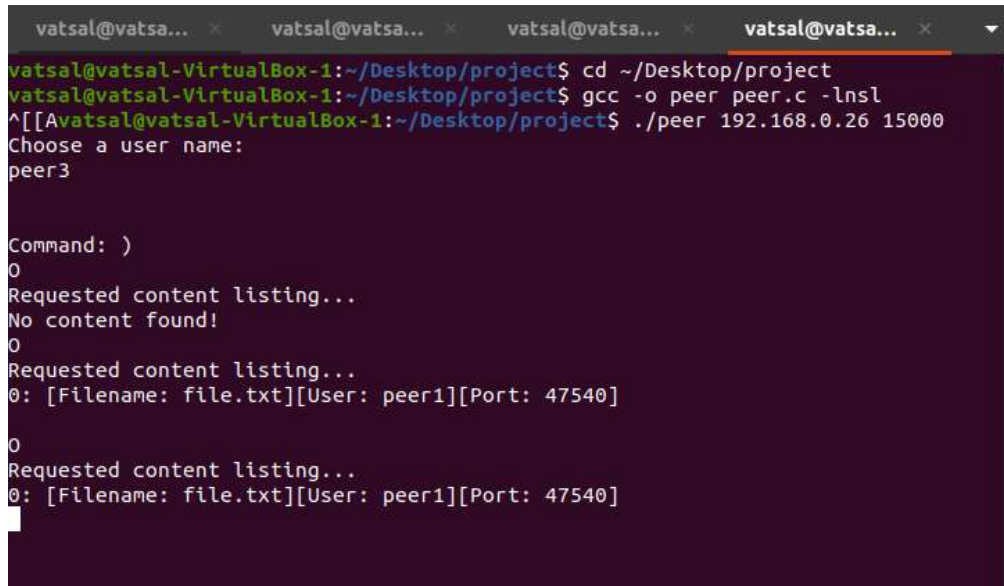
*Figure 4: D-Type Command in Terminal by Peer2*

The test was successful. A second user named peer2 could download file.txt, which was already registered to the first peer1. After registering file.txt under the changed name, the other files were registered under the original name, with no user-input reversion to the original name.



### Test 5: Third user creation

Verifying that the file has now been registered for peer1. This can be done by creating peer3 and entering the O-Type PDU.



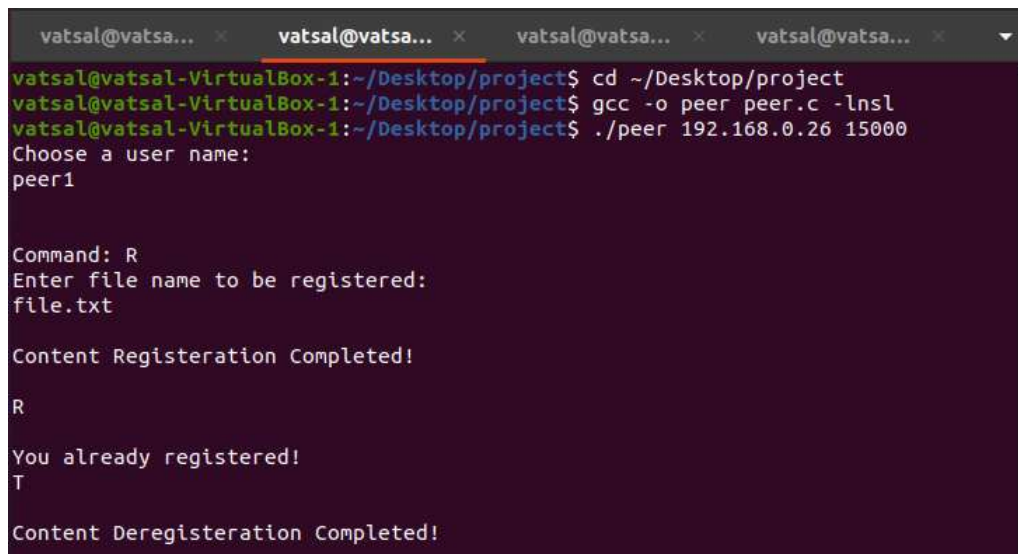
```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
^[[Avatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer3

Command: )
0
Requested content listing...
No content found!
0
Requested content listing...
0: [Filename: file.txt][User: peer1][Port: 47540]
0
Requested content listing...
0: [Filename: file.txt][User: peer1][Port: 47540]
```

*Figure 6: O-Type Command in Terminal by Peer3*

### Test 6: De-Register Files with Original Username

The user peer1 should be able to de-register his files. The file name is then requested, and when entered a T-Type PDU is sent to the index server to locate the file. That file is then deregistered.



```
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ cd ~/Desktop/project
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ gcc -o peer peer.c -lnsl
vatsal@vatsal-VirtualBox-1:~/Desktop/project$ ./peer 192.168.0.26 15000
Choose a user name:
peer1

Command: R
Enter file name to be registered:
file.txt

Content Registration Completed!

R
You already registered!
T

Content Deregistration Completed!
```

*Figure 5: T-Type Command in Terminal by Peer1*

## Conclusions

The requirements were satisfied, but not without hurdles or better ways to implement the P2P application. The task of this assignment was to create and execute a peer-to-peer file-sharing application. The elementary purpose of peer-to-index server communication and peer-to-peer communication was established. A client would be able register and de-register files that are present in the index server. A client can also request to see a list of online registries and download the given files from a peer. Moreover, the clients could also gain access to the local file directory. The clients can quit the application and upon doing so, the files that they registered with would automatically be removed.

Regardless of the achieving most of the required specifications, the tests were not entirely satisfactory. For instance, the username entered on client side was not stored for future references. The system did not identify the username of the peer, so the file registered under that substitute username could not be accessed to be deregistered. Moreover, upon entering the command for content listing ("L") for peer3, the application only printed peer1 and not peer2. On the bright side, it was noted that the files registered under the peer1 could be deregistered without any issues. In totality, the application only encountered issues since it was created to maintain files based on username and not to keep a continuous track of permanent username changes.

## References

1. Tanenbaum, A. S., and D. J. Wetherall. Computer Networks, 5th ed. Prentice Hall: Toronto, © 2010, p. 748.
2. Wikipedia. "Peer-to-Peer". Retrieved August 4 2020, from <https://en.wikipedia.org/wiki/Peer-to-peer>.

# APPENDICES

## P2Pserver Source Code

/\* P2P Server

Message types:

A - used by the server to acknowledge the success of registration

C - Content

D - download content between peers

E - Error messages from the Server

L - Location of the content server peer

Q - used by chat users for de-registration

R - used for registration

S - Search content

\*/

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <stdlib.h>

#include <string.h>

#include <netdb.h>

#include <stdio.h>

#include <arpa/inet.h>

#define MSG1 "The request content cannot be found."

#define BUFLLEN 100 /\*packet size\*/

#define NAMESIZ 10

#define MAX\_NUM\_CON 200

typedef struct entry{

char usr[NAMESIZ];

```

    struct sockaddr_in addr;

    short    token;

    struct entry *next;
} ENTRY;

```

```

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

```

```

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

```

```

void search(int, char *, struct sockaddr_in *);
void registration(int, char *, struct sockaddr_in *);
void deregistration(int, char *, struct sockaddr_in *);
int main(int argc, char *argv[])

```

```

{
    struct sockaddr_in sin, p_addr; /* client address*/
    ENTRY      *p_entry;
    char    service = "15000";      /* port number */
    char    name[NAMESIZ], usr[NAMESIZ];
    int     alen = sizeof(struct sockaddr_in); /* from-address length */
    int     s, n, i, len, p_sock; /* socket type/descriptor */
    int     pdulen=sizeof(PDU);

```

```
struct hostent      *hp;
PDU    rpdu;
struct sockaddr_in fsin;
```

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

```
switch (argc) {
case    1:
    break;

case    2:
    service = argv[1];
    break;

default:
    fprintf(stderr, "usage: chat_server [host [port]]\n");

}
```

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

```
/* Map service name to port number */
```

```
sin.sin_port = htons((u_short)atoi(service));
```

```
/* Allocate a socket */
```

```
s = socket(AF_INET, SOCK_DGRAM, 0);
if (s < 0){
```

```

        fprintf(stderr, "can't creat socket\n");
        exit(1);
    }

```

```

/* Bind the socket */

```

```

if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    fprintf(stderr, "can't bind to %s port\n", service);

```

```

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

```

```

/*      Content Registration Request      */
    if(rpdu.type == 'R'){
        registration(s, rpdu.data, &fsin);
        continue;
    }

```

```

/* Search Content      */
    if(rpdu.type == 'S'){
        search(s, rpdu.data, &fsin);
        continue;
    }

```

```

/*      List current Content */
    if(rpdu.type == 'O'){
        printf("Request to list all the Content\n");
        len = 0;
        tpdu.type = 'O';

```

```

        for(n=0; n<max_index; n++){
            if(list[n].head != NULL){
                strcpy(&tpdu.data[len], list[n].name);
                len = len + strlen(list[n].name);
                tpdu.data[len] = '\n';
                len = len+1;
            }
        }
        tpdu.data[len] = '\0';
        sendto(s, &tpdu, sizeof(tpdu), 0,(struct sockaddr *) &fsin, sizeof(fsin));
        continue;
    }

    /*      De-registration      */
    if(rpdu.type == 'T'){
        deregistration(s, rpdu.data, &fsin);
        continue;
    }

    /*      Receive unregonized PDU type */
    printf("protocol error\n");

}

return(0);
}

```

```

void search(int s, char *data, struct sockaddr_in *addr)
{
    int n,find=-1;
    char name[NAMESIZ];

```



```
ENTRY      *head, *curr;
```

```
PDU  tpdu;
```

```
strcpy(name, data);
```

```
for(n=0; n<max_index; n++){
```

```
    if((strcmp(name,list[n].name)==0) && (list[n].head!=NULL) ){
```

```
        find= n;
```

```
        break;
```

```
    }
```

```
}
```

```
if(find == -1){
```

```
    tpdu.type= 'E';
```

```
    printf("Search Unsuccessful\n");
```

```
    sendto(s, &tpdu, sizeof(tpdu),0, (struct sockaddr *)addr, sizeof(struct sockaddr));
```

```
    return;
```

```
}
```

```
else{
```

```
    head = list[find].head;
```

```
    curr = head;
```

```
    while(curr != NULL){
```

```
        if(curr->token == 1){
```

```
            printf("Search: Find peer %s port number %d\n", curr->usr,ntohs(curr->addr.sin_port));
```

```
            curr->token = 0;
```

```
            if(curr->next == NULL) head->token=1;
```

```
            else curr->next->token = 1;;
```

```
            tpdu.type = 'S';
```

```
            memcpy(tpdu.data, &curr->addr, sizeof(struct sockaddr));
```

```
            n = sendto(s, &tpdu, sizeof(tpdu),0, (struct sockaddr *)addr, sizeof(struct sockaddr));
```

```
            printf("Search: n= %d\n",n);
```

```

        return;

    }

    curr = curr->next;

}

}

}

void deregistration(int s, char *data, struct sockaddr_in *addr)
{
    ENTRY      *curr, *head, *prev;

    int      n, find=-1, findu=-1;

    char      name[NAMESIZ], usr[NAMESIZ];

    PDU      tpdu;

    printf("Entering the deregistration routine\n");

    strcpy(name, data);

    strcpy(usr, &data[NAMESIZ]);

    for(n=0; n<max_index; n++){
        if((strcmp(name,list[n].name)==0) && (list[n].head!=NULL) ){
            find= n;

            break;
        }
    }

    printf("deregistration: find %d\n",find);

    printf("deregistration: usr name %s\n", usr);

    if(find == -1){
        tpdu.type = 'E';
    }
}

```

```

        sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
        return;
    }
else{
    head = curr = prev = list[find].head;
    while(curr != NULL){
        if(strcmp(curr->usr,usr) == 0){
            findu = 1;
            printf("deregistration: find the entry\n");
            printf("deregistration: remove the entry: %d\n", ntohs(curr->addr.sin_port));
            if(curr == head)
                list[find].head = curr->next;
            else{
                prev->next = curr->next;
            }
            if(curr->token == 1){
                if(curr->next == NULL) list[find].head->token=1;
                else curr->next->token = 1;;
            }
        }
        free(curr);
        break;
    }
    prev = curr;
    curr = curr->next;
}
}

if(findu == -1)
    tpdu.type = 'E';
else{
    if(list[find].head == NULL && find == max_index-1) max_index--;
}

```

```

        tpdu.type='T';
    }
    sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));
    return;
}

```

```

void registration(int s, char *data, struct sockaddr_in *addr)

```

```

{
    ENTRY *curr, *tmp;
    int n, dup = -1, find=-1;
    int p_sock;
    char name[NAMESIZ], usr[NAMESIZ];
    struct sockaddr_in p_addr;
    PDU    tpdu;

    memcpy(&p_addr, addr, sizeof(struct sockaddr));
    strcpy(name, &data[NAMESIZ]);
    strcpy(usr, data);
    memcpy(&p_addr.sin_port, &data[NAMESIZ+NAMESIZ], sizeof(p_addr.sin_port));

    tmp = (ENTRY *) malloc(sizeof(ENTRY));

    memcpy(&tmp->addr, &p_addr, sizeof(struct sockaddr_in));
    strcpy(tmp->usr, usr);
    tmp->next = NULL;

    /* Search for the record */

    for(n=0; n<max_index; n++){
        if((strcmp(name,list[n].name)==0) && (list[n].head!=NULL)){

```

```

        find= n;
        break;
    }
}

if(find==-1){ //First peer that registers the content
    for(n=0; n<=max_index; n++){
        if(list[n].head==NULL){
            list[n].head = tmp;
            strcpy(list[n].name, name);
            tmp->token = 1;
            if(n == max_index) ++max_index;
            tpdu.type = 'A';
            printf("Registration: Content %s registered:\n",name);
            printf("Registration: Content peer name: %s\n",usr);
            printf("Registration: IP Address %s\n", inet_ntoa(p_addr.sin_addr));
            printf("registration: peer port number %d\n", ntohs(p_addr.sin_port));
            break;
        }
    }
}

sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *) addr, sizeof(struct sockaddr));
}

else{
curr = list[find].head;
while(curr != NULL){
    if(strcmp(curr->usr, usr) == 0){ // duplicated user name
        dup = 1;
        break;
    }
    else{

```

```

        curr = curr->next;
    }
}

if(dup == 1){
    free(tmp);
    tpdu.type = 'E';
    printf("Registration: Duplicate Username %s\n", usr);
}
else{
    curr = list[find].head;
    curr->token = 0;
    while(curr->next != NULL){
        curr = curr->next;
        curr->token = 0;
    }
    curr->next = tmp;
    tmp->token=1;
    tpdu.type = 'A';
    printf("Registration: Content %s registered\n",name);
    printf("Registration: Content peer name: %s\n",usr);
    printf("Registration: IP Address %s\n", inet_ntoa(p_addr.sin_addr));
    printf("registration: peer port number %d\n", ntohs(p_addr.sin_port));
}

sendto(s, &tpdu, sizeof(tpdu), 0, (struct sockaddr *)addr, sizeof(struct sockaddr));

}

return;
}

```

[P2Ppeer Source Code](#)

/\* A P2P Client

Refer to the following functions:

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

\*/

```
#include <stdio.h>
```

```
#include <netdb.h>
```

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
#include <stdlib.h>
```

```
#include <strings.h>
```

```
#include <string.h>
```

```
#include <arpa/inet.h>
```

```
#include <unistd.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
#include <signal.h>
```

```
#include <sys/select.h>
```

```
#include <errno.h>
```

```
#define QUIT "quit"
```

```
#define SERVER_PORT 10000 /* well-known port */
```

```
#define BUFLLEN      100          /* buffer length */
```



```

#define NAMESIZ 10

#define MAXCON    200

#define ERRMSG1    "Cannot find the media\n"


typedef struct
{
    char    type;
    char    data[BUFLen];
} PDU;

PDU    rpdu;


struct {
    int    val;
    char    name[NAMESIZ];
} table[MAXCON]; //Keep Track of the registered content


char usr[NAMESIZ];


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


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

```

```
void    handler();
```

```
int main(int argc, char **argv)
```

```
{
```

```
    int    s_port = SERVER_PORT;
```

```
    int    n;
```

```
    int    alen = sizeof(struct sockaddr_in);
```

```
    struct hostent      *hp;
```

```
    struct sockaddr_in server;
```

```
    char    c, name[NAMESIZ];
```

```
    char    *host = "localhost";
```

```
    struct sigaction sa;
```

```
    switch(argc){
```

```
        case 1:
```

```
            break;
```

```
        case 2:
```

```
            host = argv[1];
```

```
            break;
```

```
        case 3:
```

```
            host = argv[1];
```

```
            s_port = atoi(argv[2]);
```

```
            break;
```

```
        default:
```

```
            printf("Usage: %s host [port]\n", argv[0]);
```

```
            exit(1);
```

```
    }
```

```
/* UDP Connection with the index server
```

```
*/
```

```
    memset(&server, 0, alen);
```

```

server.sin_family = AF_INET;
server.sin_port = htons(s_port);
if ( hp = gethostbyname(host) )
    memcpy(&server.sin_addr, hp->h_addr, hp->h_length);
else if ( (server.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE ){
printf("Can't get host entry \n");
    exit(1);
}

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

if (connect(s_sock, (struct sockaddr *)&server, sizeof(server)) < 0){
    printf("Can't connect \n");
    exit(1);
}

/*      Enter User Name      */
printf("Choose a user name\n");
scanf("%s",usr);

/* Initialization of SELECT`structure and table structure */
FD_ZERO(&afds);
FD_SET(s_sock, &afds);    /* Listening on the index server socket */
FD_SET(0, &afds);    /* Listening on the read descriptor */
nfds = 1;
for(n=0; n<MAXCON; n++)
    table[n].val = -1;

```

```

/*      Setup signal handler      */
sa.sa_handler = handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGINT, &sa, NULL);

/* Main Loop */
while(1){
    printf("Command:\n");

    memcpy(&rfd, &afd, sizeof(rfd));
    if (select(nfd, &rfd, NULL, NULL, NULL) == -1){
        printf("select error: %s\n",strerror(errno));
        exit(1);
    }

    if (FD_ISSET(0, &rfd)) { /* Command from the user */
        c = getchar();

/*      Command options      */
        if(c=='?'){
            printf("R-Content Registration; T-Content Deregistration; L-List Local Content\n");
            printf("D-Download Content; O-List all the On-line Content; Q-Quit\n\n");
            continue;
        }

/*      Content Registration      */
        if(c=='R'){
            printf("Enter the name of the content\n");
            scanf("%s",name);

```

```

        registration(s_sock, name); // Register content to the index server
        continue;
    }

/*    List Content        */
    if(c == 'L'){
        local_list();
        continue;
    }

/*    List on-line Content    */
    if(c == 'O'){
        online_list(s_sock);
        continue;
    }

/*    Download Content        */
    if(c == 'D'){
        printf("Enter the name of the Content\n");
        scanf("%s", name);
        if(search_content(s_sock, name, &rpdu) == 0)
            if(client_download(name, &rpdu) == 0)
                registration(s_sock, name);
        continue;
    }

/*    Content Deregistration    */
    if(c == 'T'){
        printf("Enter the name of the Content\n");
        scanf("%s", name);

```

```

        deregistration(s_sock,name);
        continue;
    }

/*    Quit    */
    if(c == 'Q'){
        quit(s_sock);
        exit(1);
    }

}

/* Content transfer: Server to client */
    server_download(s_sock);
}
return 0;
}

void    quit(int s_sock)
{
    int    n;
    for(n=3; n<nfds; n++){
        if(table[n].val == 1)
            deregistration(s_sock,table[n].name);
    }
    return;
}

```

```

void    local_list()
{
    int n;

    printf("\nLocal Content List\n");
    for(n=3; n<nfds; n++){
        if(table[n].val == 1)
            printf("%s\n", table[n].name);
    }
    printf("\n");
    return;
}

void    online_list(int s_sock)
{
    PDU    tpdu;

    tpdu.type = 'O';
    write(s_sock, &tpdu, sizeof(tpdu));
    read(s_sock, &rpdu, sizeof(tpdu));
    printf("\nOn-line Content List:\n%s\n", rpdu.data);
    printf("\n");
    return;
}

void    server_download()
{
    int    n, i, new_sd, alen;
    struct sockaddr_in    client;

```



```

PDU    tpdu;

for(i=3; i<nfds; ++i){
    if (FD_ISSET(i, &rfdsets)) {
        new_sd = accept(i, (struct sockaddr *)&client, &alen);
        n=read(new_sd, &rpdu, sizeof(rpdu));
        if(rpdu.type=='D'){
            fd = open(table[i].name,O_RDONLY);
            if(fd >= 0){
                tpdu.type = 'C';
                n = read(fd, tpdu.data, BUFLLEN);
                write(new_sd, &tpdu, n+1);
                while((n = read(fd, tpdu.data, BUFLLEN))>0)
                    write(new_sd, tpdu.data, n);
                printf("complete transmission\n");
            }
            else{
                tpdu.type = 'E';
                write(new_sd, &tpdu, sizeof(tpdu));
            }
            close(fd);
            close(new_sd);
        }
    }
}

return;
}

```

```

int search_content(int s_sock, char *name, PDU *rpdu)
{
    int    n;
    PDU    tpdu;

    tpdu.type = 'S';
    memcpy(tpdu.data, name, NAMESIZ);
    if((n=write(s_sock, &tpdu, BUFLen+1)) < 0){
        printf("Search_content: write error\n");
        return -1;
    }
    if((n = read(s_sock, rpdu, BUFLen+1))<0){
        printf("Search_Content: read error\n");
        return -1;
    }
    if(rpdu->type == 'E'){
        printf("Search_Content: Cannot find content %s\n\n",name);
        return -1;
    }
    if(rpdu->type == 'S'){
        printf("Search: Find content\n");
        return 0;
    }
    else{
        printf("Search_Content: Protocol Error  \n");
        return -1;
    }
}

```

```

int client_download(char *name, PDU *pdu)

```

```

{
    struct  sockaddr_in p_addr;
    int     p_sock, n, fd;
    PDU     tpdu, rpdu;

    memcpy(&rpdu, pdu, sizeof(PDU));
    memcpy(&p_addr, rpdu.data, sizeof(p_addr));
    printf("Download: Peer IP Address %d\n", p_addr.sin_addr.s_addr);
    printf("Download: Peer Port Number %d\n", ntohs(p_addr.sin_port));
    if ((p_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        printf("Download: Can't creat a socket\n");
        return -1;
    }
    if (connect(p_sock, (struct sockaddr *)&p_addr, sizeof(p_addr)) == -1){
        printf("Download: Can't connect to the remote peer \n");
        return -1;
    }

    tpdu.type = 'D';
    write(p_sock, &tpdu, 1);
    fd = open(name, O_CREAT|O_RDWR|O_TRUNC, S_IRWXU);
    if(fd < 0){
        printf("Download: Can't open file\n");
        return -1;
    }

    n=read(p_sock, &rpdu, sizeof(PDU));
    if(rpdu.type == 'C'){
        write(fd, rpdu.data, n-1);
        while((n=read(p_sock, rpdu.data, BUFLLEN))>0){

```

```

        write(fd, rpdu.data, n);
    }
}
else{
    if(rpdu.type == 'E')
        printf("Download: Content is not available at the remote peer\n");
    else
        printf("Download: Protocol Error\n");
    close(fd);
    close(p_sock);
    return -1;
}
close(fd);
close(p_sock);
return 0;
}

```

```

void deregistration(int s_sock, char *name)

```

```

{
    int    n, find = -1;
    PDU    tpdu, rpdu;

    for(n=3; n<nfds; n++)
        if(strcmp(table[n].name, name)==0){
            if(table[n].val == 1)
                find = n;
        }
}

```

```

if(find == -1){
    printf("De-Registration: Content %s is not registered\n", name);
}

```

```

        return;
    }
else{
    if(table[find].val == -1){
        printf("De-Registration: Content %s is not registered\n",name);
        return;
    }
    FD_CLR(find, &afds);
    if(nfds == find+1) nfds = find;
    table[find].val = -1;
    close(find);
}

tpdu.type = 'T';
strcpy(tpdu.data, name);
strcpy(&tpdu.data[NAMESIZ], usr);
write(s_sock, &tpdu, sizeof(tpdu));
read(s_sock, &rpdu, sizeof(rpdu));
if(rpdu.type == 'T'){
    printf("De-Registration: Content %s is successfully de-registered\n", name);
    return;
}
else{
    printf("De-Registration: content %s is not registered\n",name);
    return;
}
}

void registration(int s_sock,char *name)

```

```

{

    struct sockaddr_in reg_addr, addr;

    int p_sock, alen, n;

int p_port = 0;

    PDU    tpdu, rpdu;


    alen = sizeof(struct sockaddr_in);

/* Create a stream socket for content download */
if ((p_sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
    printf("Registration: Can't creat a socket\n");
    return;
}


/* Create a socket for Content download */
memset(&reg_addr, 0, sizeof(struct sockaddr_in));
reg_addr.sin_family = AF_INET;
reg_addr.sin_port = htons(0);
reg_addr.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(p_sock, (struct sockaddr *)&reg_addr, sizeof(reg_addr)) == -1){
    printf("Registration: Can't bind name to socket\n");
    return;
}

    listen(p_sock,5);

    getsockname(p_sock, (struct sockaddr *)&reg_addr, &alen);

tpdu.type = 'R';

memcpy(&tpdu.data[NAMESIZ], name, NAMESIZ);
memcpy(&tpdu.data[2*NAMESIZ], &reg_addr.sin_port, sizeof(reg_addr.sin_port));

    while(1){
        memcpy(tpdu.data, usr, NAMESIZ);

        if( (n=write(s_sock, &tpdu, sizeof(tpdu))) <=0 ){

```

```

        printf("Registration: write Error\n");
        return;
    }
    if ((n=read(s_sock, &rpdu, sizeof(rpdu))) <0) {
        printf("Registration: read error\n");
        return;
    }
    if(rpdu.type == 'A'){
        FD_SET(p_sock, &afds);
        table[p_sock].val = 1;
        strcpy(table[p_sock].name,name);
        if(nfds <= p_sock)
            nfds = p_sock+1;
        printf("Registration: name %s\n",name);
        printf("Registration: Port: %d\n\n", ntohs(reg_addr.sin_port));
        return;
    }
    if(rpdu.type == 'E'){
        printf("Name conflict! Please choose other name for the content\n");
        scanf("%s",usr);
        if(strcmp(name, QUIT)== 0) exit(1);
    }

}

printf("Registration: protocol Error\n\n");
}

void handler()
{
    quit(s_sock);}

```