**Ryerson University**

**Department of Electrical, Computer and Biomedical Engineering**

**Computer Networks**

**Development of a Peer-to-Peer Application**

Authors

500776145 Kosigan Shanmugarajah

T.A: Hamzah Qadeer

Professor: Bobby Ma

December 4, 2020

**Contents**

[**Introduction** 3](#_Toc57907744)

[**Objective** 3](#_Toc57907745)

[**Basic Socket Programming Background Description** 3](#_Toc57907746)

[**Socket Programming Connections; server and client** 3](#_Toc57907747)

[**Code Design and Implementation** 5](#_Toc57907748)

[**Server and Client Protocols Descriptions** 5](#_Toc57907749)

[**Server and Client Program Implementations** 8](#_Toc57907750)

[**Observations and Results** 9](#_Toc57907751)

[**Content Registration** 9](#_Toc57907752)

[**Content Download** 10](#_Toc57907753)

[**Content De-Registration** 11](#_Toc57907754)

[**Conclusions** 12](#_Toc57907755)

[**References** 13](#_Toc57907756)

[**Appendix** 14](#_Toc57907757)

# **Introduction**

## **Objective**

The purpose of this project is to implement a peer-to-peer application in which peers can exchange the contents between themselves through the support of an index server. A peer can be both a content server of a set of contents or a content client of a set of contents depending on the task it wishes to accomplish. If a peer contains a content ready for download by other peers, it acts as a content server of that content, while if it wishes to download a content, then it acts as a content client of a set of contents. The content server’s role is to register the contents to the index server, and the content clients will search the address of that content from the index server, for downloading. Upon finding, it can download the contents. In addition, the communication between an index server and a peer, is established based on UDP while content downloads are established based on TCP connections.

## **Basic Socket Programming Background Description**

Socket programming provides services required for communicating between applications in a local system or a wide TCP based network. It’s implementation in this project is a key factor aiding in establishing unique TCP connections between peers for clear and efficient communication. Since a socket is bound to a port number, these unique connections are possible. In that the port number is used by the TCP layer to identify the application where contents are specifically destined to be sent. This destination is the sockets endpoint, which is a combination of IP address and the port number. Doing such makes it possible for unique connections to be established between peers, allowing for fast and smooth content exchanges.

## **Socket Programming Connections, server and client**

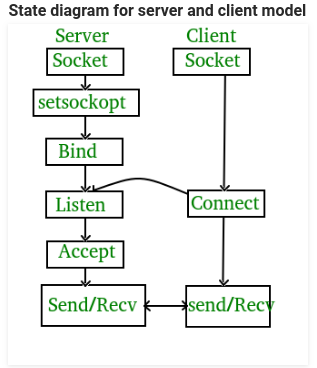


Figure 1. State Diagram for Server and Client

Stages for Server:

* Socket creation: int sockfd = socket( domain, type, protocol)
* Sockfd: socket descriptor, an integer
* Domain: integer, communication domain. (Ex, AF\_INET [IPv4 protocol])
* Type: communication type SOCK\_STREAM: TCP(reliable, connection oriented), SOCK\_DGRAM:UDP(unreliable, connectionless)
* Protocol: value for Internet Protocol(IP) which is 0. This is the same number which shows on protocol field in the IP header of a packet.
* Setsockopt: int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen). This will help in getting rid of options for the socket referred by the file descriptor sockfd.
* Bind: int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)
* Listen: int listen sockfd, int backlog
* Accept: int new\_socket = accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen

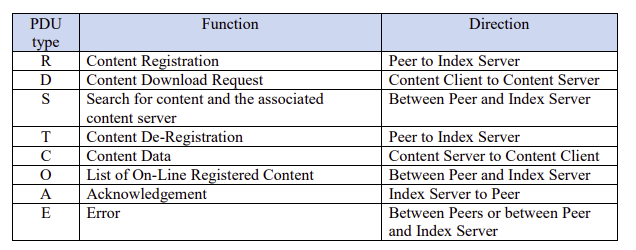
Stages for Client:

Socket connection: Same as server’s socket creation.

* Connect: int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)
* FD\_ZERO: zeros out the file descriptor
* FD\_SET: adds a socket to a list of valid sockets
* Select: Allows you to keep track of multiple file descriptors
* FD\_ISSET: Checks if the socket provide is being communicated with

# **Code Design and Implementation**

## **Server and Client Protocols Descriptions**

Table 1: The eight PDU types implemented in the application.

The data field of all PDU types excluding C-type PDU, has a maximum byte size of 100 bytes. Since the C-type PDU serves for carrying the contents, its data size is the size of the content being carried.

R-type PDU:

By sending an R-type PDU a peer is able to register its contents to the index server. The data portion of the PDU includes the peer name, content name, and the address of the content server for that particular content, as shown in the figure below.

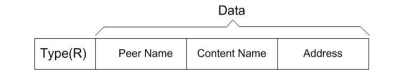


Figure 2. R-type PDU, data format

Upon receiving an R-type PDU from a peer, the index server will check whether another peer with the same name has already registered the same content. If so, then the index server will send E-type PDU and prompt the peer to choose a different peer name. If the peer name is unique, the content server will register the content and address associated with it and send back an A-type PDU to notify the successful registration of the content. As shown in the figure below.

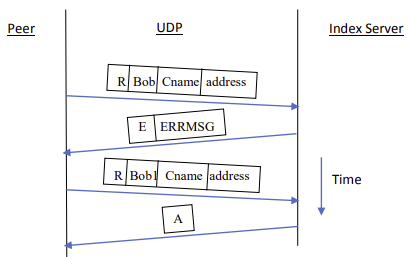


Figure 3. Content Registration when a peer is prompted to choose a different peer name.

D-type PDU:

A peer can request for a content download by contacting the index server to search for the address of the corresponding content server. This is done by sending an S-type PDU with the format shown in the figure below.



Figure 4. S-type PDU, data format

If the content requested for download is available, the index server will respond with an S-type PDU and search for the content. If the content doesn’t exist, the index server will simply send back an, error, E-type PDU.

Upon receiving an S-type PDU from the index server in a case where the requested content is available, the peer will extract the address present in the PDU and proceed onto setting up a unique TCP connection with the content server. If the TCP connection is successfully established, the peer will send a D-type PDU to the content server to commence the download. Upon the completion of the download and receiving the D-type PDU, the content server will then send the content over through the C-type PDU, which contains the contents requested. After successfully downloading the contents the peer is prompted to register the received contents to the index server. Figure below illustrates the transactions for the D-type PDU.

Note that for efficiency, during the search process a dequeue is implemented in order to ensure that the index server chooses the latest content server (the content server that has been used the least). This is done to prevent the index server from overloading, keeping the load distributed evenly.

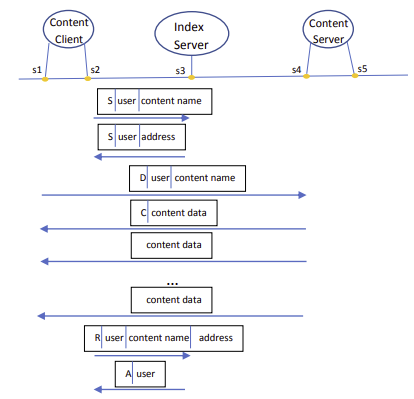


Figure 5. D-type PDU: content search, download, and content registration. i.e. s2, s3 and s5 are UDP sockets while s1 and s4 are TCP sockets. Content client uses s2 for searching and registering content and s1 for downloading the content.

O-type PDU:

The user is able to find out all of the registered contents, and under which address it is by sending an O-type PDU to the index server. Upon receiving the request, the server will respond with an O-type PDU which includes the list of all online registered contents.

T-type PDU:

If a peer wishes to de-register a content, it can do so, by sending a T-type PDU to the index server. If the content that the peer wishes to de-register is available, the index server will de-register it and send back an A-type PDU, to notify the peer the successful de-registration of the content requested. If the content doesn’t exist, it will send back an E-type PDU and prompt the peer to choose a different content name.

Quitting

If a peer decides quit, it needs to de-register all of its registered contents, by individually sending multiple T-type PDU requests, until no more content is available for de-registration.

## **Server and Client Program Implementations**

**Server:**

arr\_content\_servers: this data structure is used to stored all registered content

After every ‘R’-type sent by the client the server stores content name, peer name and port in the content servers’ array. After a ‘S’-type is sent by the client, then find\_content\_server() is called to check if the content is exist. Once the it can be confirmed that the content exist, the server will send back the content server struct that contains the port to connect the content server.

remove\_content(), will be used when the client sends a ‘T’-type to the server. This function will remove the given pdu from the array of content servers.

dequeue(), will send the most recently download content server to the end of the list. This will then allow for the user to download from the most recently download content client. The advantage of this is that you can get the latest and most update record of a file. Also, if There’s some kind of corruption with the file it can be easily be tracked.

find\_content\_server(), this function will find if the content requested actually exist. Otherwise the server will send an error back to the client.

find\_matching\_peer\_name\_content\_server(), this function will determine whether an existing combination of the peer name and content that is trying to be registered already exists.

For ‘O’-type, the server sends back the array of content server. The client will read the array and parse it according to make it more readable for the client.

**Client:**

create\_tcp\_connection(), this functions creates a new tcp connection for every new content that is being registered. When creating a new socket the socket type is SOCK\_STREAM, which used for TCP connections. htons(0), is used to set the socket to a random port which is then returned and stored in the pdu that is sent to the server. After the port and address have been set, it is important to bind that struct to an open socket. This socket will always be listening for any incoming tcp connections.

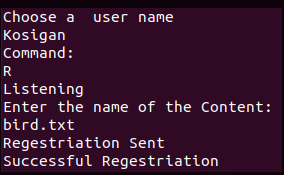
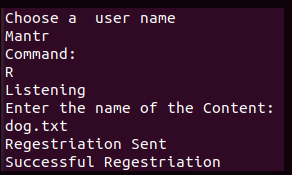
If the client is reading from the terminal, then it will trigger with a 0 socket value. Which is where FD\_ISSET will catch it and proceed to read from the terminal for commands or any other instructions.

The client sends the a ‘D’-type with the port number provide by the server, and using FD\_ISSET the client will realize that another socket is trying to communicate with it and accept the socket connection. The content server will then the data in 100-byte sizes as a ‘C’-type.

# **Observations and Results**

After implementing the application, the code was compiled and each PDU type was tested. Figures below show the successful results obtained when commanding each PDU types on the terminal.

## **Content Registration**

****

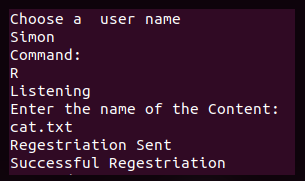
****

Figure 6. Three unique peers with one unique content associated with each, registered.

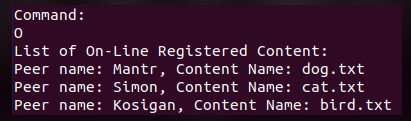
****

Figure 7. List of all online registered contents, with associated addresses.

The above two figures illustrate the successful content registration of a unique content associated with a unique peer.

## **Content Download**

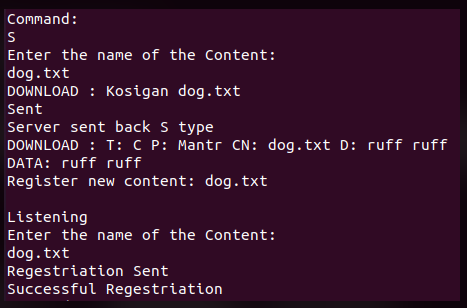
****

Figure 8. Peer “Kosigan” downloads the “dog.txt” file from peer “Mantr” and registers the newly obtained content.

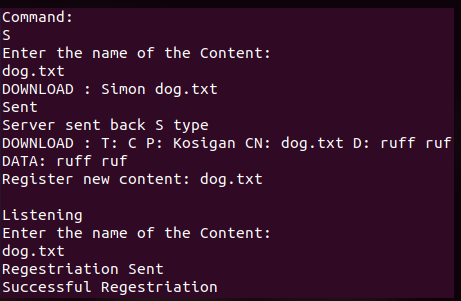
****

Figure 9. Peer “Simon” downloads the “dog.txt” file from peer “Kosigan” and registers the newly obtained content.

From Figure 9, it can be seen that peer “Simon” downloaded the content from peer “Kosigan” and not peer “Mantr”, this is because of the implementation of the dequeue method. Doing such allowed for the index server to search and download from the least used content server.

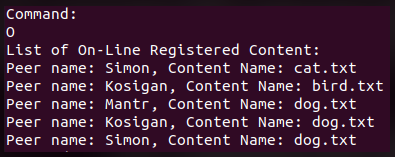
****

Figure 10. List of all online registered contents, inclduing the updated ones.

## **Content De-Registration**

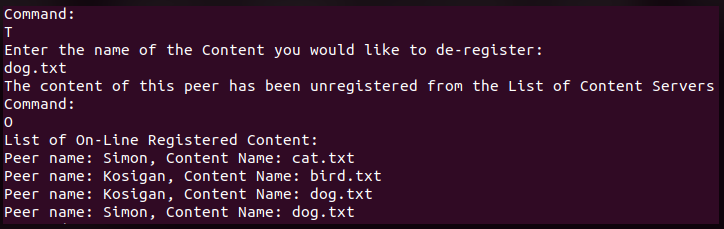


Figure 11. De-regestering the contents of peer “Mantr”, and lisitng all of the online registered contents.

Note how de-regestering the content of peer “Mantr” removed it from the online registered content list. This is because peer “Mantr” had only one content, thus, removing that content, would meaning it no longer has any available content to download, and thus the peer has quitted. If the peer requests for a content, and registeres the content, it will be displayed on the list of online registered content, and will be active again.

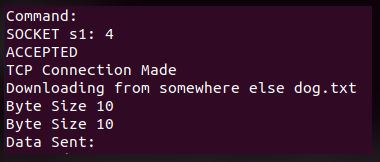


Figure 12. Demonstrates the byte size fixed to 10 bytes for the peer name and content name fields for time efficiency.

# **Conclusions**

In conclusion the peer-to-peer application was successfully implemented. As per the criteria, the application established a UDP connection between the index server and the peer, and a TCP connection between the peers. The application was able to handle all eight required PDU types and use it accordingly to get a task accomplished. i.e. content registration to the index server, content download from a content server (successfully searching for the content form the least used server and sending back either an acknowledgment or an error depending on the availability), content de-registration, and listing all of the online registered content. Incorporation of the socket programming allowed for unique TCP connections to be established, which resulted in effective communication amongst the peer with faster and clear delivery of the requested contents, if available. Overall, a working peer-to-peer application was implemented that handled and performed all of the required PDU types.

# **References**

**[1]** COE 768 Project 1 Manual, “*Peer-to-peer Application*”. Ryerson University. Dec 2, 2019. [Online]. Available: Ryerson D2L portal.

# **Appendix**

**Server.c**

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <netdb.h>

#include <stdio.h>

#include <time.h>

#include <stdbool.h>

/\*------------------------------------------------------------------------

\* main - Peer-by-Peer Applications

\*------------------------------------------------------------------------

\*/

#define BUFLEN 256

int

main(int argc, char \*argv[])

{

struct sockaddr\_in fsin; /\* the from address of a client \*/

char buf[100],list[1000]; /\* "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, tag; /\* socket descriptor and socket type \*/

int port=3000;

struct pdu{

char type;

char peer\_name[10];

char content\_name[10];

unsigned short port;

char data[100];

};

struct content\_server{

char peer\_name[10];

char content\_name[10];

unsigned short port;

};

struct content\_server arr\_content\_servers[10];

struct content\_server desired\_content\_server;

int n, r, bytes\_to\_read, index = 0;

int words = 0,byteSize, bytes\_written;

char c, \*foo;

char fileName[BUFLEN];

switch(argc){

case 1:

break;

case 2:

port = atoi(argv[1]);

break;

default:

fprintf(stderr, "Usage: %s [port]\n", argv[0]);

exit(1);

}

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

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = INADDR\_ANY;

sin.sin\_port = htons(port);

/\* Allocate a socket \*/

s = socket(AF\_INET, SOCK\_DGRAM, 0);

if (s < 0)

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

/\* Bind the socket \*/

if (bind(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)

fprintf(stderr, "can't bind to %d port\n",port);

listen(s, 5);

alen = sizeof(fsin);

int remove\_content(struct pdu given)

{

for(int k=0;k<10;k++)

{

if (strcmp(arr\_content\_servers[k].peer\_name, given.peer\_name) == 0 && strcmp(arr\_content\_servers[k].content\_name, given.content\_name) == 0)

{

memset(arr\_content\_servers[k].peer\_name, 0, sizeof arr\_content\_servers[k].peer\_name);

memset(arr\_content\_servers[k].content\_name, 0, sizeof arr\_content\_servers[k].content\_name);

arr\_content\_servers[k].port = 0;

return 1;

}

}

return 0;

}

void dequeue(int tag)

{

arr\_content\_servers[index] = arr\_content\_servers[tag];

index++;

memset(arr\_content\_servers[tag].peer\_name, 0, sizeof arr\_content\_servers[tag].peer\_name);

memset(arr\_content\_servers[tag].content\_name, 0, sizeof arr\_content\_servers[tag].content\_name);

arr\_content\_servers[tag].port = 0;

}

int find\_content\_server(struct pdu given)

{

for(int n=0;n<10;n++)

{

if (strcmp(arr\_content\_servers[n].content\_name, given.content\_name)==0)

{

desired\_content\_server = arr\_content\_servers[n];

tag = n;

return 1;

}

}

return 0;

}

int find\_matching\_peer\_name\_content\_server(struct pdu given)

{

for(int r=0;r<10;r++)

{

if (strcmp(arr\_content\_servers[r].peer\_name, given.peer\_name)==0 && strcmp(arr\_content\_servers[r].content\_name, given.content\_name)==0)

{

printf("Peer: %s already registered this content: %s\n",arr\_content\_servers[r].peer\_name, arr\_content\_servers[r].content\_name);

return 1;

}

}

return 0;

}

while (1) {

struct pdu spdu,rpdu;

//server obtains fileanme

if (recvfrom(s, &rpdu, sizeof(rpdu), 0,

(struct sockaddr \*)&fsin, &alen) < 0)

fprintf(stderr, "recvfrom error\n");

FILE \*f;

struct content\_server \*bp;

if(rpdu.type == 'R')

{

printf("Regestriation Begins\n");

if(find\_matching\_peer\_name\_content\_server (rpdu))

{

spdu.type='E';

sprintf(spdu.data,"This peer is already registerd");

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

else

{

strcpy(arr\_content\_servers[index].peer\_name, rpdu.peer\_name);

strcpy(arr\_content\_servers[index].content\_name, rpdu.content\_name);

arr\_content\_servers[index].port = rpdu.port;

index++;

spdu.type='A'; //Acknowledgement //Peer create open TCP socket client side

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

}

if (rpdu.type == 'S') //Search Request

{

printf("Search Request Begin\n");

if(!find\_content\_server(rpdu))

{

printf("Error is happening \n");

spdu.type='E';

sprintf(spdu.data,"A peer with this content name can not be found");

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

else

{

printf("Send Port to Client\n");

spdu.type = 'S';

strcpy(spdu.content\_name, desired\_content\_server.content\_name);

strcpy(spdu.peer\_name, desired\_content\_server.peer\_name);

spdu.port = desired\_content\_server.port;

printf("Desired Content Server %d \n",desired\_content\_server.port);

dequeue(tag);

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

}

if(rpdu.type == 'O')

{

spdu.type='O';

bp=arr\_content\_servers;

sendto(s, (struct content\_server\*)&arr\_content\_servers, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

if (rpdu.type == 'T')

{

printf("Removing Content from List for Peer \n ");

if (remove\_content(rpdu))

{

printf("Acknowledge the peer is deleted \n");

spdu.type='A'; //Acknowledgement //Peer has been removed from content server list

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

else

{

spdu.type='E';

sendto(s, (struct pdu\*)&spdu, 1000, 0,

(struct sockaddr \*)&fsin, sizeof(fsin));

}

}

}

}

**Client.c**

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#define BUFSIZE 64

#define BUFLEN 100

#define MSG "Any Message \n"

/\*------------------------------------------------------------------------

\* main - UDP client for TIME service that prints the resulting time

\*------------------------------------------------------------------------

\*/

int

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

{

char \*host = "localhost";

struct hostent \*phe; /\* pointer to host information entry \*/

struct sockaddr\_in sin,fsin,client,num\_fd, sock\_addr, content\_server; /\* an Internet endpoint address \*/

int s, s1, s2, sock, n, type, new\_sd, client\_len; /\* socket descriptor and socket type \*/

switch (argc) {

case 1:

break;

case 2:

host = argv[1];

case 3:

host = argv[1];

port = atoi(argv[2]);

break;

default:

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

exit(1);

}

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

sin.sin\_family = AF\_INET;

sin.sin\_port = htons(port);

/\* Map host name to IP address, allowing for dotted decimal \*/

if ( phe = gethostbyname(host) ){

memcpy(&sin.sin\_addr, phe->h\_addr, phe->h\_length);

}

else if ( (sin.sin\_addr.s\_addr = inet\_addr(host)) == INADDR\_NONE )

fprintf(stderr, "Can't get host entry \n");

/\* Allocate a socket \*/

s = socket(AF\_INET, SOCK\_DGRAM, 0);

if (s < 0)

fprintf(stderr, "Can't create socket \n");

/\* Connect the socket \*/

if (connect(s, (struct sockaddr \*)&sin, sizeof(sin)) < 0)

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

//create struct

char command;

char filename[10],username[10];

struct pdu{

char type;

char peer\_name[10];

char content\_name[10];

unsigned short port;

char data[101];

};

struct content\_server{

char peer\_name[10];

char content\_name[10];

unsigned short port;

};

struct content\_server content\_list[10];

int bytes\_read,alen,alen1, i;

unsigned short myPort;

fd\_set rfds, afds;

struct pdu rpdu,spdu,tpdu;

char cmd [100],buf3[500],buf[BUFLEN];

FILE \*fp, \*fp1;

char \*bp;

int download,num\_port,byteSize, bytes\_written;

struct sockaddr\_in reg\_addr;

num\_port = 0;

unsigned short create\_tcp\_connection()

{

s1 = socket(AF\_INET,SOCK\_STREAM, 0);

/\* Create a stream socket \*/

if ( s1 == -1) {

fprintf(stderr, "Can't creat a socket\n");

exit(1);

}

reg\_addr.sin\_family = AF\_INET;

reg\_addr.sin\_port = htons(0);

reg\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if(bind(s1, (struct sockaddr \*)&reg\_addr, sizeof(reg\_addr)) == -1)

{

printf("Can't bind name to to socket");

exit(1);

}

alen1 = sizeof(struct sockaddr\_in);

getsockname(s1,(struct sockaddr\*)&reg\_addr,&alen1);

myPort = reg\_addr.sin\_port;

FD\_SET(s1,&afds);

if(listen(s1, 5) == 0)

{

printf("Listening\n");

}

else

{

printf("Not Listening\n");

}

num\_port++;

return myPort;

}

//ask user for download or exit

printf("Choose a user name\n");

n=read(0,username,100);

username[n-1]='\0';

if (10 > sizeof(spdu.peer\_name))

{

printf("Choose an appropriate user name less than 10 bytes\n");

exit(1);

}

//create forever loop for multiple downloads

while(1){

printf("Command: \n");

FD\_ZERO(&afds);

FD\_SET(s1,&afds);

FD\_SET(0,&afds);

memcpy(&rfds,&afds,sizeof(rfds));

if(select(FD\_SETSIZE,&rfds, NULL, NULL, NULL) < 0)

{

exit(1);

}

if(FD\_ISSET(0,&rfds))

{

scanf(" %c",&command);

if (command == 'S')

{

spdu.type='S';

printf("Enter the name of the Content:\n");

n=read(0,spdu.content\_name, 100);

strcpy(spdu.peer\_name,username);

spdu.content\_name[n-1]='\0'; //make filename char string

printf("DOWNLOAD : %s %s \n",spdu.peer\_name,spdu.content\_name);

write(s, &spdu, 1000);

printf("Sent \n");

n=read(s, &rpdu, 1000);

if ( rpdu.type == 'E')

{

printf("Error \n");

}

else if( rpdu.type == 'S')

{

printf("Server sent back S type \n");

tpdu.type='D';

strcpy(tpdu.peer\_name,rpdu.peer\_name);

strcpy(tpdu.content\_name,rpdu.content\_name);

memset(&content\_server, 0, sizeof(content\_server));

content\_server.sin\_family = AF\_INET;

content\_server.sin\_port = rpdu.port;

s2 = socket(AF\_INET,SOCK\_STREAM, 0);

content\_server.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

/\* Connect the socket \*/

if (connect(s2, (struct sockaddr \*)&content\_server, sizeof(content\_server)) < 0)

{

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

exit(1);

}

write(s2, &tpdu, 1000);

n=read(s2, &rpdu,200);

printf("DOWNLOAD : T: %c P: %s CN: %s D: %s \n",rpdu.type, rpdu.peer\_name, rpdu.content\_name, rpdu.data);

if(rpdu.type == 'C'){

fp1 =fopen(rpdu.content\_name,"w");

while(rpdu.type == 'C'){

rpdu.data[100] = '\0';

printf("DATA: %s",rpdu.data);

fprintf(fp1, "%s",rpdu.data);

n = read(s2,&rpdu,200);

}

}

fclose(fp1);

close(s2);

// Send registration for new content

printf("\n");

printf("Register new content: %s \n",tpdu.content\_name);

printf("\n");

command = 'R';

}

}

if (command == 'R')

{

spdu.type='R';

spdu.port = create\_tcp\_connection();

printf("Enter the name of the Content:\n");

n=read(0,spdu.content\_name, 100);

strcpy(spdu.peer\_name,username);

spdu.content\_name[n-1]='\0'; //make filename char string

write(s, &spdu, 1000);

printf("Regestriation Sent\n");

n=read(s, &rpdu, 1000);

if( rpdu.type == 'A')

{

printf("Successful Regestriation\n");

}

else if( rpdu.type == 'E')

{

printf("Regestriation Error \n");

}

}

if (command == 'O')

{

spdu.type='O';

write(s, &spdu, 1000);

n=read(s, &content\_list, 1000);

printf("List of On-Line Registered Content: \n");

for(int i=0;i<10;i++)

{

if(strlen(content\_list[i].peer\_name) > 1)

{

printf("Peer name: %s, Content Name: %s \n", content\_list[i].peer\_name, content\_list[i].content\_name);

}

}

}

if (command == 'T')

{

spdu.type='T';

printf("Enter the name of the Content you would like to de-register:\n");

n=read(0,spdu.content\_name, 100);

spdu.content\_name[n-1]='\0';

strcpy(spdu.peer\_name,username);

write(s, &spdu, 1000);

n=read(s, &rpdu, 1000);

if (rpdu.type == 'A')

{

printf("The content of this peer has been unregistered from the List of Content Servers\n");

}

else

{

printf("Peer was not unregistered\n");

}

}

}

if(FD\_ISSET(s1,&rfds))

{

printf("SOCKET s1: %d \n", s1);

alen = sizeof(client);

new\_sd = accept(s1, (struct sockaddr \*)&client,&alen);

if(new\_sd == -1)

{

printf("DID NOT ACCEPT \n");

}

else

{

printf("ACCEPTED \n");

}

printf("TCP Connection Made \n");

n=read(new\_sd,&rpdu,1000);

if(rpdu.type == 'D'){

printf("Downloading from somewhere else %s \n", rpdu.content\_name);

fp = fopen(rpdu.content\_name, "r");

if(fp == NULL){

printf("File not found in peer \n");

}else

{

fseek(fp,0L, SEEK\_END);

byteSize = ftell(fp);

rewind(fp);

printf("Byte Size %d \n", byteSize);

fgets(buf3, byteSize, fp);

bp = buf3;

rpdu.type = 'C';

strcpy(rpdu.peer\_name,username);

while(byteSize>0){

printf("Byte Size %d \n", byteSize);

printf("Data Sent: %s \n",rpdu.data);

strcpy(rpdu.data,bp);

rpdu.type = 'C';

bytes\_written = write(new\_sd, &rpdu, 200);

byteSize -=100;

bp +=100; //go to the next 100 bytes

}

rpdu.type = 'A';

write(new\_sd, &rpdu, 101);

close(new\_sd);

}

}

}

}

}