

Faculty of Engineering and Architectural Science

# Department of Electrical and Computer Engineering

|  |  |
| --- | --- |
| **Course Number** | COE 768 |
| **Course Title** | Computer Networks |
| **Semester/Year** | 7th Semester/4th Year |
| **Lab No** | 6 |
| **Instructor Name** | Bobby Ma |
| **Section No** | 2 |

|  |  |
| --- | --- |
| **Submission Date** | 12/04/2020 |
| **Due Date** | 12/04/2020 |

|  |  |
| --- | --- |
| **Name** | **Signature\*** |
|  |  |
|  |  |
|  |  |

\*By signing above, you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result

in a “0” on the work, an “F” in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at:

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

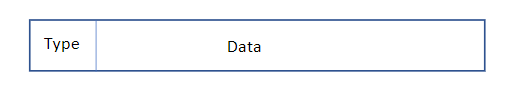
**Introduction**

This lab project allowed for implementation and creation of a P2P (Peer-to-Peer) application. This application consists of several peers communicating through an index server allowing them to exchange content among themselves. Peers that have a piece of content (text files, audio files, or videos) that is available for download by the other peers is known as the content server of that content. Peers that want to download a piece of content are called the content clients of that content. Peers can be both a content server of a piece content and a content client of another set of content. The content server will register its content to the index server. This results in the content client finding the address of a content server from the index server. The communication between the index server and a peer is based on UDP (User Datagram Protocol) while the content download is based on TCP (Transmission Control Protocol).

TCP and UDP are network protocols that can be used to transfer data over to a server and is a form of socket programming. Socket programming connects two nodes that are on the same network allowing for communication between them. One node will listen at a port, while another will try to form a connection. The server will act as the listener, while the client will reach out.

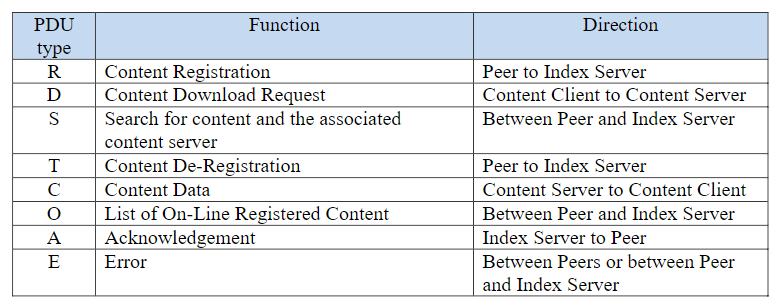
**Description**

The data that will be sent between the client and index servers will be imposed by a protocol data unit (PDU) as seen in figure 1.



**Figure 1: PDU format used**

The PDU type will determine the function, while it also carries any necessary data. Table 1 summarizes the functions for each PDU type and the direction which the PDU moves.

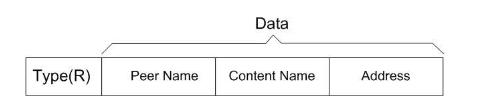


**Table 1: Summary of PDU Functions**

The communication between a peer and the index server is based on UDP. The index server provides services of content registration, deregistration, and search. In addition, it will respond with the list of registered content upon the request from a peer.

**Content Registration**

Content must first me registered to the index server by a peer through sending an R-type PDU. R-Type PDU’s consist of the peer name of the one registering, the content name, and the address of the content server for that content. The format can be seen in figure 2.

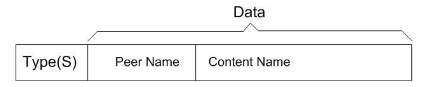


**Figure 2: R-Type PDU Format**

When the index server receives an R-type PDU, it will first check if another peer with the same name has registered the same content. If this happens, the index server will send an E-type PDU to prompt the peer to choose another peer name. If there is no conflict of peer name, the content server will register the content and store the associated address. Subsequently, it sends back an A-type PDU to acknowledge the success of the content registration. Note that before sending a registration request, a peer must first create a TCP socket for content download. The port number associated with the socket is part of the address registered to the index server. If a peer registers more than one piece of content, it should open a separate TCP socket for each piece of content.

**Content Download**

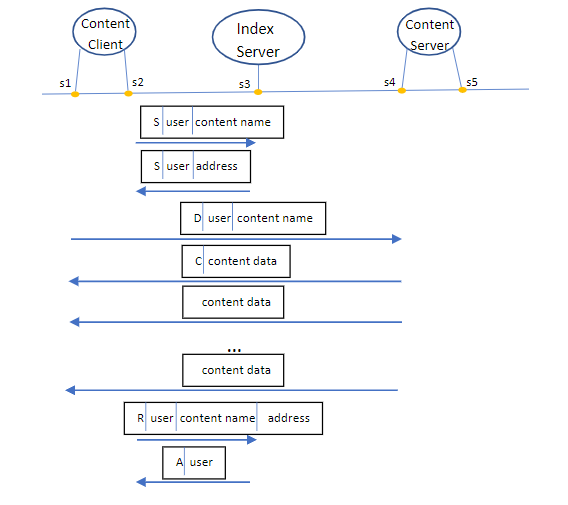
To download a piece of content, a peer first contacts the index server to look for the address of the corresponding content server through sending an S-Type PDU. S-type PDU’s consist of the peer name and the content name. The format can be seen in figure 3.



**Figure 3: S-Type PDU Format**

If the content does not exist, it will respond with an E-type PDU which is the error type. If the content exists, the index server will respond with an S-type PDU that contains the address of the content server and setup a TCP connection with the content server. Once the TCP connection is established, the peer sends a D-type PDU to the content server which will trigger the download. The content server will receive the D-type PDU, and if the content is available, will send it back through a C-type PDU that contains the content. If the content is too large to send in one packet, the PDU can be broken into smaller packets. Once all the data is sent back to the peer, the TCP connection will be terminated, and the peer will be notified.

After the content has been downloaded, the peer will register the content to the index server, therefore becoming the server of the content. The index server can have multiple servers for one piece of content. To distribute it evenly, once a search request is received, the index server will the content server that has been used the least. The process can be seen in figure 4;



**Figure 4: Content Search, Downloading, and Registering**

**Content Listing**

To get a list of all registered contents, the user can send an O-type PDU to the index server. The index server will send back all registered contents from each peer.

**Content De-Registration**

The application also provides peers the ability to de-register their registered content through sending a T-type PDU. If the content exists and was successfully deregistered, the index server will send back an A-type PDU, acknowledging the deregistration.

**Quit**

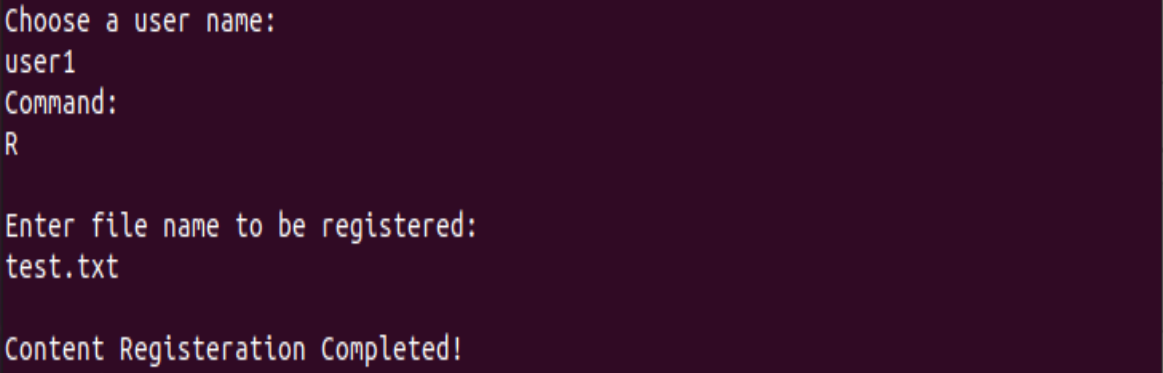
When a peer wishes to quit, they must first de-register all its registered content. This is so the content registration information of the index server will stay up to date. This can be accomplished by sending a series of T-type PDUs to the index server before quitting.

**Observations & Analysis**

The process for the application involves sets of commands within the terminal which delivers a PDU to the index server where it will then respond accordingly. The commands are as followed:

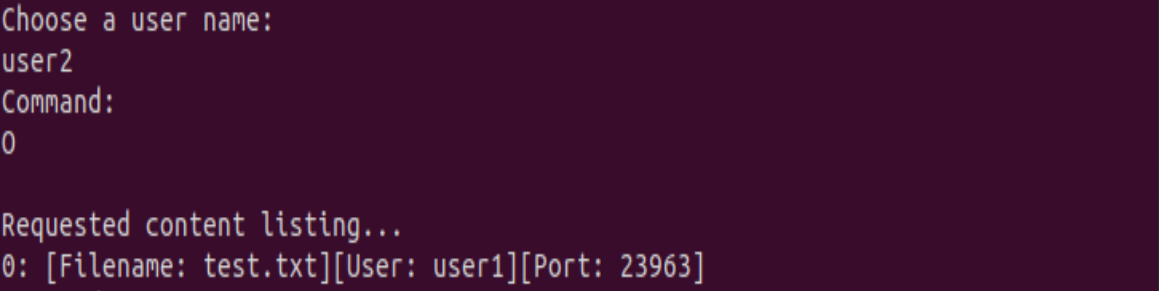
* R – Register a file
* O – List the registered contents and their information
* S – Search for the address of a specific piece of content
* D – Download the content searched for
* Q – Exit the Application

The program begins with the user inputting a username for themselves known as user1. This name is stored within an array in the index server. Content must first be registered to run any other commands, so an R is entered into the system. The file name is specified and an R-Type PDU is sent to the index server. If the file exists an A-Type PDU is returned to acknowledge the registration as seen in Figure 5.



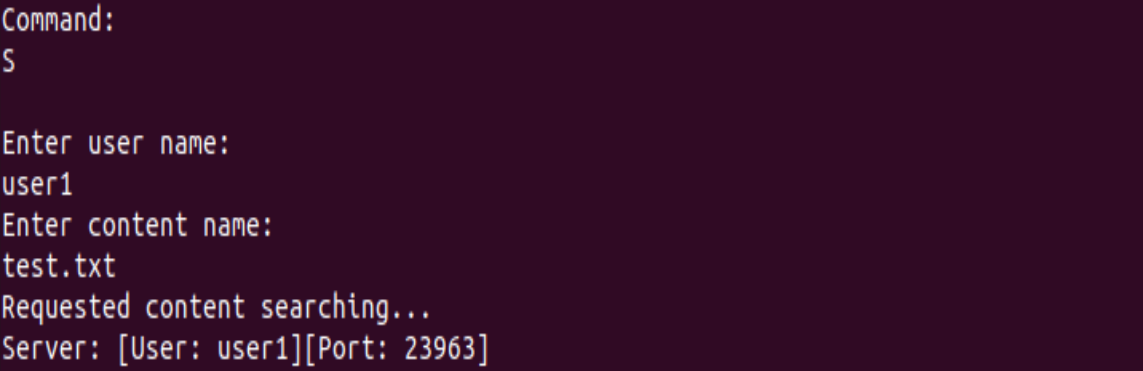
**Figure 5: R-Type Command in Terminal by User1**

To verify the existence of the registered file, a second user is registered known as user2. This user is also stored within the index server. An O command is entered into the system, which sends an O-Type PDU to the index server. The server retrieves all the registered content, and its information, and sends it back to the user as seen in figure 6.



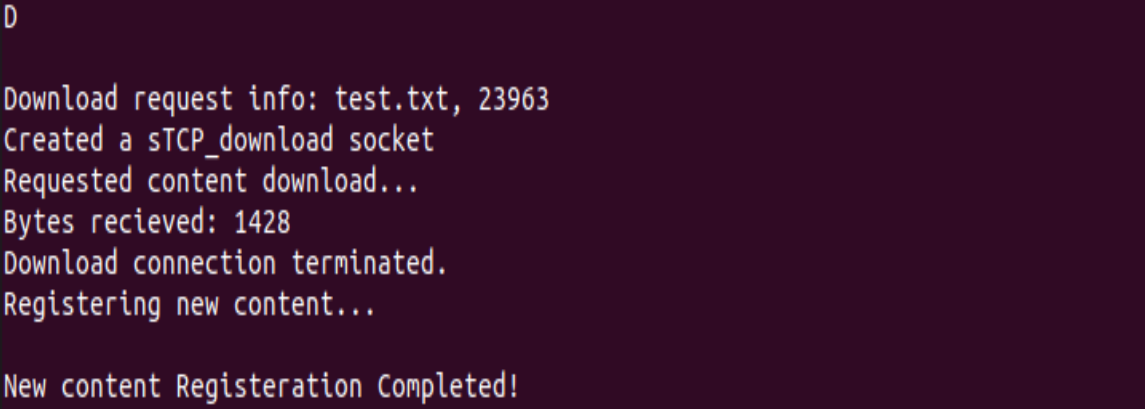
**Figure 6: O-Type Command in Terminal by User2**

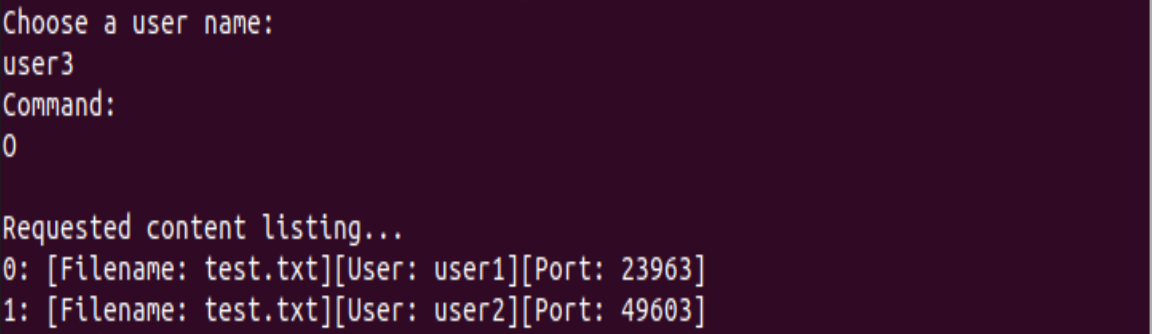
Before the user can download the file, the address of the file must first be found. An S is entered which requests to search for a specific file by inputting the username as well as the content name. If the content exists, an S-Type PDU is returned with the address and set up a TCP connection as seen in Figure 7.



**Figure 7: S-Type Command in Terminal by User2**

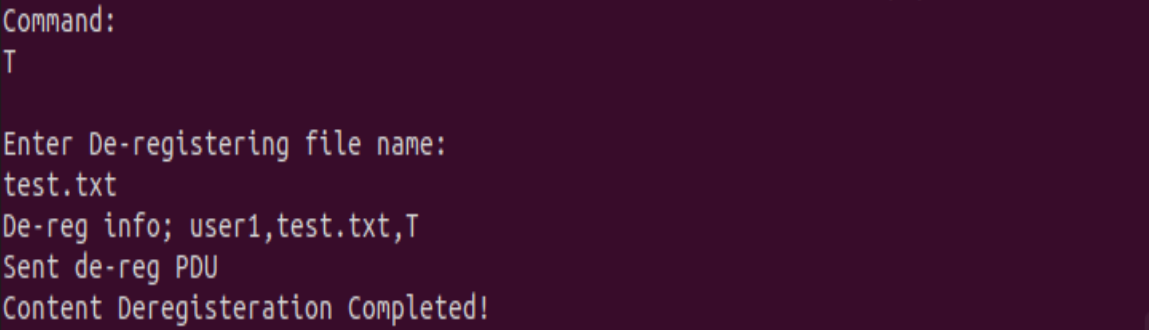
With the connection, the user can now download the content by entering a D. This will send the D-Type PDU to request for the data to be downloaded. The server will respond with a C-Type PDU which holds all the data of the file. Due to size limitations of a TCP packet, data is sent with a maximum of 1460 bytes. If the file is larger, the packet will split into a number of smaller packets until the entire file is sent. Once the file is downloaded, the file will be automatically registered back into the index server. This results in user2 now being a content server for the file. This can be seen in figure 8.

**Figure 8: D-Type Command in Terminal by User2**

To verify that the file has now been registered for user2, a third user knows as user3 was created. An O was entered to display the list of registered contents which now display the same file for both user1 and user2 held in different ports, as seen in figure 9.

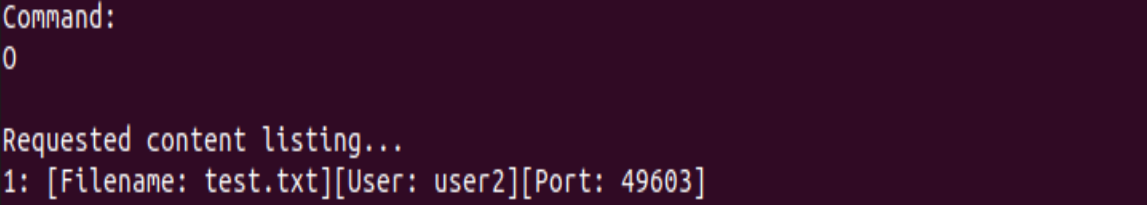
**Figure 9: O-Type Command in Terminal by User3**

Content can then be deregistered by user1 entering a T into the terminal. The file name is then requested, and when entered a T-Type PDU is sent to the index server to locate the file. If it exists, that file is then deregistered, and an acknowledgement is sent to the peer as seen in figure 10.

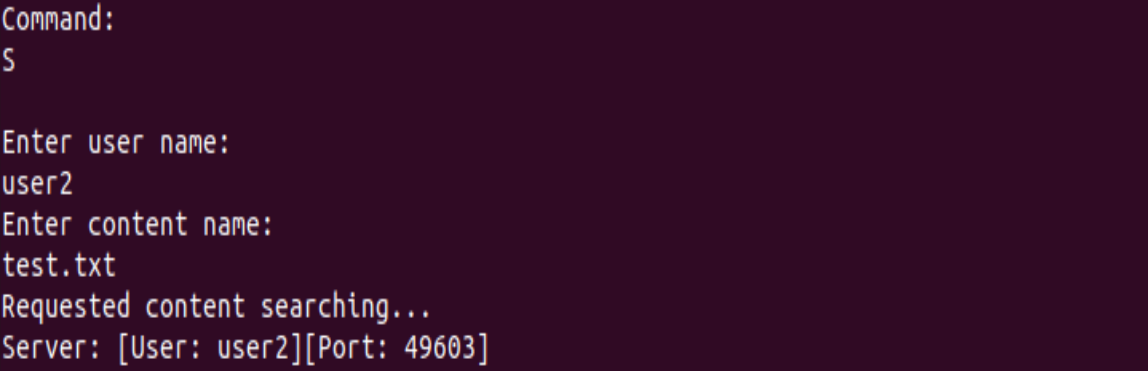


**Figure 10: T-Type Command in Terminal by User1**

To verify the deregistration, user3 enters the O command again to see the list of registered content. Although both pieces of content contained the same name, the T command only removes the content registered for the user who enters it. The file “test.txt” was deregistered for user1, but not for user2 as seen in figure 11.

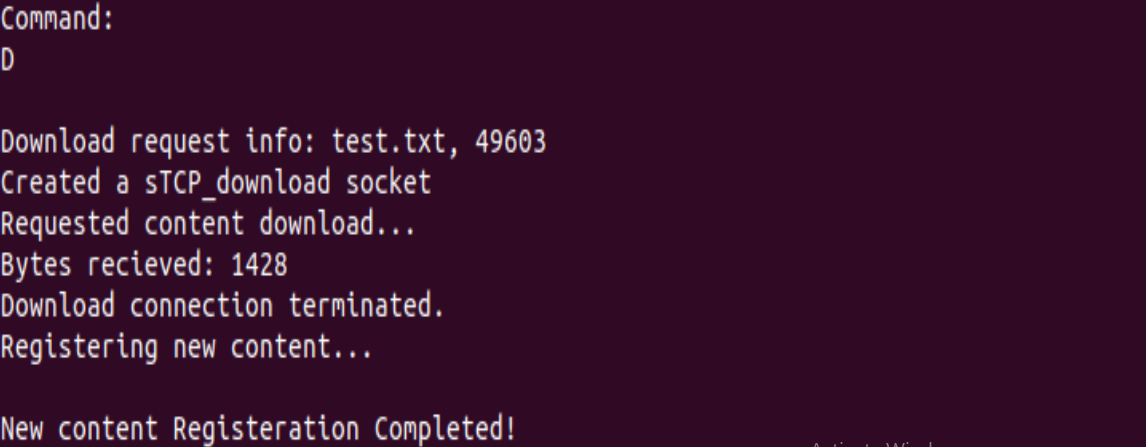


**Figure 11: O-Type Command in Terminal by User3**

The third user can then confirm the registered content from user2 by requesting it as well. This is done the same as before by first entering an S and the username and content name as seen in figure 12.

**Figure 12: S-Type Command in Terminal by User3**

The download is then done by entering a D, which will download the content and then allow user 3 to register the file and serve as a content register themselves as seen in figure 13



**Figure 13: D-Type Command in Terminal by User3**

**Conclusions**

In conclusion, the goal of the lab was to develop and implement a Peer to Peer application which consists of an index server and various peers. Using the C language, the index server and peers were developed and applied to the application using Linux. Information sent between the index server and the peer, is based on UDP, while when downloading content, it is based on a TCP connection. The index server acted as a platform for users to register pieces of content, allowing them to be shared among other users. Through registering the content, the user then serves as a content server, and with the index server users can find the addresses of the content servers to then download the content. Overall, from the terminal and from analyzing the packets with Wireshark, communication of information over a network can be visualized.

**References**

[1] COE768 Project: P2P Application. Ryerson University. Dec 3, 2020. Retrieved from: https://courses.ryerson.ca/d2l/le/content/391241/viewContent/3032068/View

**Appendix 1: Index Server**#include <unistd.h>

#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>

#define DBUSER 5

#define DBSIZE 5

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

\* main - Iterative UDP server for TIME service

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

\*/

struct contentreg{

char type;

char user [10];

char contentName[10];

struct sockaddr\_in address;

char data[10];

};

struct contentDest{

char type;

char user [10];

char contentName[10];

struct sockaddr\_in address;

char data[100];

};

struct error{

char type;

char data [100];

};

struct pdu{

char type;

char data[100];

int cutoff;

};

struct UserRegistry{

char type;

char user [10];

char contentName[DBSIZE][10];

struct sockaddr\_in address;

char data[10];

};

struct contentreg peers[DBUSER];

struct UserRegistry userDir[DBUSER];

int checkUser(struct contentreg userReg) {

for(int i = 0; i < DBUSER; i ++ ) {

if (strcmp(userDir[i].user,userReg.user)==0) {

printf("Check user: %s with %s\n", userReg.user,userDir[i].user);

return 1; //match

}

}

return 0;

}

int checkContent(struct contentreg userReg) {

int eval = 0 ;

for(int i = 0; i < DBUSER; i ++ ) {

if (strcmp(userReg.user,userDir[i].user)==0) {

for (int j =0; j< DBSIZE; j++) {

eval = strcmp(userDir[i].contentName[j],userReg.contentName);

if (eval == 0) {

return 1; //match

}

}

}

}

return 0;

}

struct searchResult {

struct UserRegistry catalog[DBUSER];

int total;

};

int checkUserSearch(struct searchResult results, struct contentreg userReg) {

int eval = 0 ;

for(int i = 0; i < DBUSER; i ++ ) {

eval = strcmp(userReg.user,results.catalog[i].user);

if (eval == 0) {

return 1;

}

}

return 0;

}

void removeContent(struct contentreg userReg) {

for(int i = 0; i < DBUSER; i ++ ) {

if (strcmp(userReg.user,userDir[i].user) == 0) {

for (int j =0; j< DBSIZE; j++) {

if (strcmp(userReg.contentName,userDir[i].contentName[j]) == 0){

strcpy(userDir[i].contentName[j],"");

break;

}

}

break;

}

}

}

void removeUser(struct contentreg user) {

for(int i = 0; i < DBUSER; i ++ ) {

if (strcmp(userDir[i].user,user.user) == 0) {

strcpy(userDir[i].user, "");

for (int j =0; j< DBSIZE; j++) {

strcpy(userDir[i].contentName[j],"");

}

break;

}

}

}

struct searchResult listContents() {

struct searchResult results;

int usrCount = 0;

int cnCount = 0;

printf("\nListing all contents...\n");

for (int i=0;i<DBUSER;i++) {

if (userDir[i].user != NULL && strcmp(userDir[i].user, "")!=0) {

printf("Registered user: %s\n", userDir[i].user);

results.catalog[usrCount] = userDir[i];

for (int j=0;j<DBSIZE; j++) {

if (userDir[i].contentName[j] != NULL && strcmp(userDir[i].contentName[j], "")!=0) {

printf("--Registered content: %s\n", userDir[i].contentName[j]);

strcpy(results.catalog[usrCount].contentName[j], userDir[i].contentName[j]);

cnCount++;

}

}

usrCount++;

}

}

results.total = cnCount;

return results;

}

struct contentreg searchContent(char user[10], char contentName [10]){

struct contentreg result;

printf("\nSearching for contents from user: %s\n", user);

result.type = 0;

for (int i=0;i<DBUSER;i++) {

if ((strcmp(userDir[i].user, user) == 0)) {

strcpy(result.user, userDir[i].user);

result.address = userDir[i].address;

for (int j=0;j<DBSIZE; j++) {

if ((strcmp(userDir[i].contentName[j], contentName) == 0)) {

strcpy(result.contentName, userDir[i].contentName[j]);

printf("[user: %s] [%d] - content: %s\n",result.user,j, result.contentName);

result.type = 1;

break;

}

}

}

}

return result;

}

void registerUser (struct contentreg user) {

for (int i = 0;i <DBUSER; i++){

if (userDir[i].user == NULL || strcmp(userDir[i].user, "")==0 || strcmp(userDir[i].user, user.user)==0) {

strcpy(userDir[i].user, user.user);

userDir[i].address = user.address;

printf("Registered user: %s\n", userDir[i].user);

printf("Registered address port: %d\n", userDir[i].address.sin\_port);

for (int j = 0; j < DBSIZE; j++) {

if (userDir[i].contentName[j] == NULL || strcmp(userDir[i].contentName[j], "") == 0) {

strcpy(userDir[i].contentName[j], user.contentName);

printf("Registered Content: %s\n",userDir[i].contentName[j]);

break;

}

}

break;

}

}

}

void deRegisterUser (struct contentreg user) {

if (checkUser(user)) {

removeUser(user);

}

}

void deRegisterContent(struct contentreg user) {

if (checkContent(user)) {

removeContent(user);

}

}

int

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

{

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

char buf[100], file[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;

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, DBSIZE);

alen = sizeof(fsin);

int i, n, nread, j = 0, peerCount = 0;

struct contentreg receivedReg;

struct error message;

struct pdu serverT, serverO, client;

struct contentDest serverS;

char type1 ='R';

while (1) {

n = recvfrom(s, &receivedReg, sizeof(receivedReg), 0,

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

if(n<0)

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

switch(receivedReg.type){

case 'R':

{

if (peerCount == DBUSER) {

message.type ='E';

strcpy(message.data, "Max number of users registered. Please try another time!\n");

sendto(s, &message, sizeof(message), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

}

if(checkContent(receivedReg) == 1 && checkUser(receivedReg) == 1) {

message.type = 'E';

strcpy(message.data, "Content already exists of the same user! Please pick a different filename: \n");

sendto(s, &message, sizeof(message), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

break;

}

else{

message.type ='A';

registerUser(receivedReg);

printf("The file %s is registered to port %d\n",receivedReg.contentName, receivedReg.address.sin\_port);

sendto(s, &message, sizeof(message), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

peerCount++;

break;

}

break;

}

case 'T':{

struct contentreg result = searchContent(receivedReg.user, receivedReg.contentName);

char returnMsg[100] = "";

printf("De-reg: name: %s user: %s", receivedReg.contentName, receivedReg.user);

fflush(stdout);

if (result.type != 1) {

message.type = 'E';

strcpy(serverT.data, "Requested content not found!");

}

else {

removeContent(receivedReg);

peerCount--;

message.type = 'A';

strcpy(message.data, "Content has been deregistered!");

sendto(s, &message, sizeof(message), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

break;

}

sendto(s, &serverT, sizeof(serverT), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

fflush(stdout);

break;

}

case 'O':

{

char returnMsg[100] = "";

struct searchResult results = listContents();

int pollcounter = 0;

if (results.total == 0) {

serverO.type = 0;

strcpy(serverO.data, "No content found!");

sendto(s, &serverO, sizeof(serverO), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

}

else {

serverO.type = 1; //start polling

printf("Search result total: %d\n", results.total);

for (int i=0;i<results.total;i++) {

if (strlen(results.catalog[i].user) > 0 && strcmp(results.catalog[i].user, "")!= 0) {

for (int j=0;j<DBSIZE; j++) {

if(strlen(results.catalog[i].contentName[j]) > 0 && strcmp(results.catalog[i].contentName[j], "")!= 0 ) {

sprintf(serverO.data,"%d: [Filename: %s][User: %s][Port: %d]",i, results.catalog[i].contentName[j], results.catalog[i].user, results.catalog[i].address.sin\_port);

//strcpy(serverO.data, returnMsg);

pollcounter++;

if (pollcounter == results.total) serverO.type = 0;//stop polling

sendto(s, &serverO, sizeof(serverO), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

if (serverO.type == 0) {

break;

}

}

}

}

if (pollcounter >= results.total) {

pollcounter = 0;

break;

}

}

}

fflush(stdout);

break;

}

case 'S':{

char returnMsg[100] = "";

printf("Requested Username %s Content Name %s \n", receivedReg.user, receivedReg.contentName);

struct contentreg result = searchContent(receivedReg.user, receivedReg.contentName);

if (checkContent(receivedReg)==0 || checkUser(receivedReg)==0 || result.type != 1) {

serverS.type == 'E';

strcpy(serverS.data, "Requested content not found!");

sendto(s, &serverS, sizeof(serverS), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

}

else {

serverS.type = 'S';

sprintf(returnMsg, "Server: [User: %s][Port: %d]", result.user, result.address.sin\_port);

strcpy(serverS.data, returnMsg);

strcpy(serverS.user, result.user);

strcpy(serverS.contentName, result.contentName);

serverS.address = result.address;

//sprintf(returnMsg, " Sent: %d: [Filename: %s][User: %s][Port: %d]",i, serverS.contentName, serverS.user, serverS.address.sin\_port);

sendto(s, &serverS, sizeof(serverS), 0,(struct sockaddr \*)&fsin, sizeof(fsin));

}

fflush(stdout);

break;

}

case 'Q': {

printf("De-registering user: %s\n", receivedReg.user);

deRegisterUser(receivedReg);

break;

}

default:{

printf("Failure!");

break;

}

}

}

}

**Appendix 2: Peer Server**

/\* time\_client.c - main \*/

#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 MSG "Any Message \n"

struct pdu{

char type;

char data[100];

int cutoff;

};

struct pduDL{

char type;

char data[1460];

int cutoff;

};

struct contentReg{

char type;

char user [10];

char contentName[10];

struct sockaddr\_in address;

char data [100];

};

struct error{

char type;

char data [100];

};

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

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

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

\*/

int

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

{

char \*host = "localhost";

int port = 3000;

char now[100], rPacket[100]; /\* 32-bit integer to hold time \*/

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

struct sockaddr\_in sin, tcpSockAddr; /\* an Internet endpoint address \*/

int sUDP,sTCP\_ul, sTCP\_dl, n, type, tcpAlen; /\* 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");

/\* Create a TCP socket for file transfer\*/

if ((sTCP\_ul = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

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

exit(1);

}

/\* Allocate a UDP socket for Index Server\*/

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

if (sUDP < 0)

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

/\* Connect the UDP socket \*/

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

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

struct contentReg peer, downloadRequest;

struct error message;

int input, alen, i, peerRegistrated =0;

char response;

struct pdu client, server;

fd\_set rfds, afds;

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

scanf(" %s", peer.user);

//printf("%s", peer.user);

//fflush(stdout);

peer.user[strlen(peer.user)] = '\0';

printf("\nCommand: ");

while(1){

//int sock = socket(AF\_INET, SOCK\_STREAM,0);

FD\_ZERO(&afds);

FD\_SET(sTCP\_ul,&afds);

FD\_SET(0,&afds);

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

select(FD\_SETSIZE, &rfds, NULL, NULL, NULL);

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

scanf(" %c", &response);

switch(response){

case '?':

{

printf("R- Content Registeration; T-Content Deregisteration; C-Download content\nD -Download Content Request; O-List all the On-line Content; Q-Quit\n");

break;

}

case 'R':

{

if (peerRegistrated == 1) {

printf("You already registered!");

break;

} /\* Create a TCP socket for file transfer\*/

if ((sTCP\_ul = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

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

exit(1);

}

/\* Bind an address to the socket \*/

bzero((char \*)&tcpSockAddr, sizeof(struct sockaddr\_in));

tcpSockAddr.sin\_family = AF\_INET;

tcpSockAddr.sin\_port = htons(0); // random port

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

if (bind(sTCP\_ul, (struct sockaddr \*)&tcpSockAddr, sizeof(tcpSockAddr)) == -1){

fprintf(stderr, "Can't bind name to socket\n");

exit(1);

}

/\* queue up to 2 connect requests \*/

listen(sTCP\_ul, 2);

tcpAlen = sizeof(struct sockaddr\_in);

getsockname(sTCP\_ul, (struct sockaddr\*) &peer.address, &tcpAlen);

//peer.address = tcpSockAddr;

peer.type = 'R';

printf("Enter file name to be registered: \n");

n=read(0,peer.contentName,10);

peer.contentName[n-1] = '\0';

(void)write(sUDP, &peer,sizeof(peer));

while((i=read(sUDP, &message, sizeof(message)))>0){

if(message.type == 'E'){

printf("Server: %s\n", message.data);

scanf(" %s", peer.user);

printf("Enter file name: \n");

scanf(" %s", peer.contentName);

peer.type = 'R';

write(sUDP, &peer, sizeof(peer));

}

else if (message.type == 'A')

{

printf("Content Registeration Completed!");

peerRegistrated = 1;

break;

}

}

break;

}

case 'T':

{

if(peerRegistrated == 0){

printf("No Files Registered!");

break;

}

peer.type = 'T';

(void)write(sUDP, &peer,sizeof(peer));

while((i=read(sUDP, &message, sizeof(message)))>0){

if(message.type == 'E'){

printf("Server: %s\n", message.data);

}

else if (message.type == 'A')

{

printf("Content Deregisteration Completed!");

peerRegistrated = 0;

break;

}

}

break;

}

case 'O':

{

printf("Requested content listing...\n");

client.type = 'O';

(void) write(sUDP, &client, 100);

while((i=read(sUDP, &server, sizeof(server)))>0){

printf("%s\n", server.data);

if (server.type == 0) break;

}

if(i<0)

printf("Failed to read file or recieve data.\n");

break;

}

case 'S':{

peer.type = 'S';

printf(" Enter user name: \n");

n=read(0, peer.user, 11);

peer.user[n-1] = '\0';

printf(" Enter content name: \n");

n=read(0, peer.contentName, 11);

peer.contentName[n-1] = '\0';

printf("Requested content searching...\n");

(void) write(sUDP, &peer, 100);

while((i=read(sUDP, &downloadRequest, sizeof(downloadRequest)))>0){

printf("%s\n", downloadRequest.data);

break;

}

printf("Download request info: %s, %d\n", downloadRequest.contentName, downloadRequest.address.sin\_port);

if(i<0)

printf("Failed to read file or recieve data.\n");

break;

}

case 'D':{

FILE \*fp;

struct pduDL buffer;

int d;

printf("Download request info: %s, %d\n", downloadRequest.contentName, downloadRequest.address.sin\_port);

/\* Create a TCP socket for file transfer\*/

if ((sTCP\_dl = socket(AF\_INET, SOCK\_STREAM, 0)) == -1) {

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

printf("Can't create a sTCP\_dl socket\n");

exit(1);

}

printf("Created a sTCP\_dl socket\n");

fflush(stdout);

if (connect(sTCP\_dl, (struct sockaddr \*)&downloadRequest.address, sizeof(downloadRequest.address)) == -1){

fprintf(stderr, "Can't bind name to sTCP\_dl socket\n");

printf("Can't bind name to sTCP\_dl socket\n");

fflush(stdout);

exit(1);

}

printf("Requested content download...\n");

(void) write(sTCP\_dl, &downloadRequest, sizeof(downloadRequest));

fp=fopen(downloadRequest.contentName, "w");

while((d = read(sTCP\_dl,&buffer, (int) sizeof(buffer)))>0){

printf("Bytes recieved: %d\n", d);

if (buffer.type == 'C') fwrite(buffer.data,d,1,fp);

else {

printf("Error: Failed during download\n");

break;

}

}

if(d<0)

printf("\nFailed to read file or recieve data.\n");

fclose(fp);

close(sTCP\_dl);

printf("Download connection terminated.\n");

}

case 'Q':{

break;

}

}

}

if(FD\_ISSET(sTCP\_ul, &rfds)){

int new\_sd = accept(sTCP\_ul, (struct sockaddr \*)&tcpSockAddr, (int\*)sizeof(tcpSockAddr));

//if (new\_sd < 0) printf("Failed to accept connection\n");

if (new\_sd > 0) {

printf("Start uploading...\n");

int BUFLEN = 10;

char buffer[BUFLEN], requestedFileName[BUFLEN], file[100], msg[BUFLEN];

char errorMsg[10] = "!";

int n, bytes\_to\_read, i, nread;

struct contentReg downloadResponse;

n = read(new\_sd,&downloadResponse, sizeof(downloadResponse));

fflush(stdout);

sprintf(requestedFileName,"%s", downloadResponse.contentName);

strtok(requestedFileName, "\n");

FILE \*f = fopen(("%s",requestedFileName), "r");

if (f == NULL) {

write(new\_sd, errorMsg, BUFLEN);

perror("Error");

}

else {

//Count number of bytes of the file

fseek(f, 0, SEEK\_END); //Moves file pointer to the last byte of the file

unsigned long fileSize = ftell(f); // Store the position of the FilePointer

fseek(f, 0, SEEK\_SET); //Moves FilePointer back to the beginning of file

unsigned long sizeCounter = 0;

unsigned long chunkSize = 1460;

while(1) {

//printf("Chunk read.\n");

sizeCounter += 1460;

//Get the last chunk size precisely

if(sizeCounter > fileSize)

chunkSize = fileSize + 1460 - sizeCounter;

nread = fread(file,1,chunkSize,f);

if (nread>0){

write(new\_sd, file, chunkSize);

printf("Chunk sent. Pointer position: %lu. Chunk Size: %lu\n",ftell(f), chunkSize);

}

else {

printf("EOF\n");

break;

}

}

fclose(f);

}

}

close(new\_sd);

}

}

exit(0);

}