CEN 502: COMPUTER SYSTEMS II  
  
A DOCUMENTATION ON  
  
P2P FILE SHARING SYSTEM

Group Number: 32

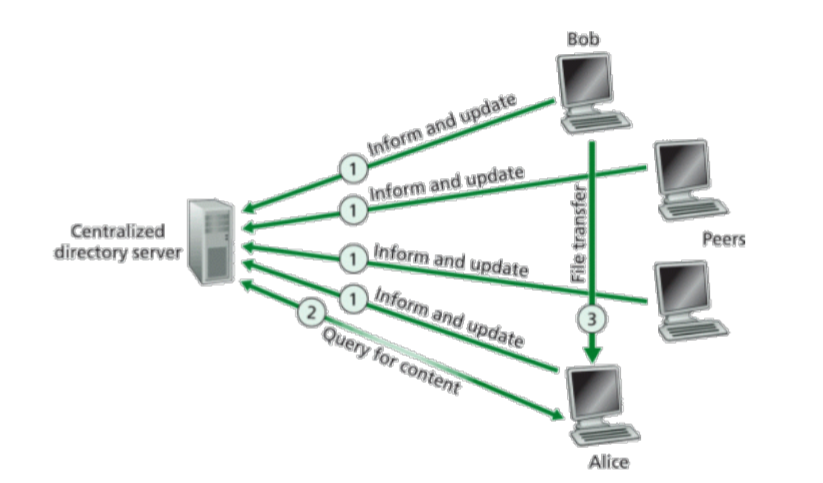
Sandesh Shetty (1209395990)   
Mohit Nangare (1209409068)

**INDEX**

|  |  |  |
| --- | --- | --- |
| Sr. No. | Topic | Page No. |
| 1 | Design | 3 |
| 2 | Message Format and Protocols | 5 |
| 3 | Pseudo code | 7 |
| 4 | Data Structures, Design Choices | 18 |
| 5 | User Guide | 20 |
| 6 | References | 21 |

**DESIGN**

Peer-To-Peer file sharing has become popular for many kinds of resource location and distribution applications including file sharing, distributed computation, multi-media messaging and content distribution. Peer-To-Peer approaches also have significant potential for supporting large scale, decentralized service oriented computing.



The P2P File Sharing System which we implemented is a classic example of Centralized P2P System (above figure) as it implements the communication between a Directory Server, P2P Client and a P2P Transient Server. The Design of the P2P File Sharing System can be easily understood if we understand the Design and Functionalities of the following classes:

* ServerUDP
* Client\_UDP
* Listen
* ClientConnection
* PeerConnection

**ServerUDP:**

This class contains various design functionalities of the Server. Here, the ServerUDP acts like Directory Server of the Centralized P2P System. Whenever a Client interacts with the Server for the first time, the Server has a UDP Socket which listens on a fixed port and then it creates a unique thread called ClientConnection which then interacts with the Client henceforth. Directory Server maintains the information like filename, file size and the host where it is present. It also replies to the queries of the various Clients. When the outgoing packet size is more than 128Bytes, the packet is fragmented and then transferred, hence implementing the Fragmentation Logic.

**Client\_UDP:**

Client\_UDP contains various design functionalities of the Client. It displays a //GUI and takes the user inputs. As per the user inputs, it calls the “method()”function which performs an activity like “Inform and Update”, “Query for Content” and “Exit”. We experience the case of Fragmentation here too, when the InformandUpdate packet is more than MTU i.e. 128Bytes. When the “Exit” is implemented, the ClientConnection thread at the server which is unique to a particular client is closed and all its entries in the main Directory Server are deleted and terminates the link between the Client and the Directory Server. On instantiation of the Client\_UDP, Listen thread is created along with the main thread of the Client\_UDP.

**Listen:**

Listen class is mainly designed for the listening functionalities. It has a ServerSocket which listens on a fixed port number. It creates Sockets and a Thread PeerConnection for every new connection request from a P2P Client.

**ClientConnection:**

The ClientConnection is a thread which is unique for each Client. The Client communicates with the Server through this thread. The packet either fragmented or original is received here in the “run()” function. On reading the request message from the client side, the “run()” function calls the specific method i.e. InformandUpdate, QueryforContent and Exit. InformandUpdate is a synchronized method, so when a thread is executing this function all other threads wanting to update the file of the Directory Server go into the “wait()” state until the current thread issues a “notify()” command, after that the next thread can execute the InformandUpdate method. This is the inherent functionality achieved on using the keyword *synchronized* in the following fashion:  
**public** **synchronized** **void** informAndUpdate(){…}

**PeerConnection:**

The PeerConnection has a Socket which is created by a Listen class and is passed to it when a thread is created for a connection request from a P2P Client. It is also unique for each P2P Client. Listen class creates this thread. This thread is used to transfer a file from a P2P transient Server to P2P Client.

**MESSAGE FORMAT and PROTOCOLS**

In this project, we have implemented both UDP and TCP. Communication between Client and Directory Server was achieved by UDP and TCP was used to transfer files from P2P Transient Server to P2P Client.

**UDP:**

Message format which was implemented is as follows:

**SequenceNumberMethodname/Hostname/IPAddress\r\n**

In case of Inform and Update, the message format will be as follows:

**1Inform and Update/Mohit\_Nangare/192.168.0.203\r\n**

For other user inputs, the message format remains same except for the Methodname which varies.

Also, we have implemented Acknowledgement Message formats and it is as follows:

**SequenceNumber:StatusCode/StatusMessage\r\n**

If the message was received the Server sends the following acknowledgement:

**1:200/OK/\r\n**

If the message was not received the Server sends the following acknowledgement:

**1:400/Unable to service the request!\r\n**

**TCP:**

Message format which was implemented is as follows:

**GET/Filename/HTTP/1.1\r\n**

Now, if you want a file say a.txt, the message format will be as follows:

**GET/a.txt/HTTP/1.1\r\n**

Also, we have implemented Acknowledgement Message formats and it is as follows:

**/StatusCode/StatusMessage\r\n**

If the message was received the Server sends the following acknowledgement:

**/200/OK/\r\n**

If the message was not received the Server sends the following acknowledgement:

**/400/Bad Request/\r\n**

If the file was not found then the Server sends the following acknowledgement:

**/404/Not Found/\r\n**

If the HTTP Version is not supported then the Server sends the following acknowledgement:

**/505/HTTP Version Not Supported/\r\n**

**PSEUDO CODE**

* **UDP Client class** (Client\_UDP.java)

1. Main method invoked on instantiation:

**public** **static** **void** main(String[] args)**throws** IOException {

**try** {

(**new** Client\_UDP()).start();

*l*=**new** Listen();//Listen.java

Thread t = **new** Thread(*l*); // Thread created which continuously   
 listens on a TCP socket through a fixed port number.

t.start();

}

**catch** (IOException e) {

e.printStackTrace();

}

}

1. run() method continuously displays a menu as a user interface to interact with the directory server and waits for the user input for further action.

System.***out***.println("Choose a Method:");

System.***out***.println("1.Inform and Update");

System.***out***.println("2.Query For Content");

System.***out***.println("3.Exit");

System.***out***.println("4.Request for a file");

Scanner i = **new** Scanner(System.***in***);

**int** input = i.nextInt();

1. Based on the input it creates a message format and calls the appropriate function to handle the request at the client side.

Ex: If user input is 1.

**switch**(input){

**case** 1:{

**try**{

message.delete(0, message.length());

message.append("Inform and Update/"+hostname+"/192.168.0.174/\r\n");

String line = in.readLine();

**while**(line != **null**){

message.append(line+"\r\n");

line = in.readLine();

System.***out***.println(message);

}

message.append("\r\n");

in.close();

method();

}**catch**(FileNotFoundException e){

System.***err***.println("Cannot open the file");

}

**break**;

}

1. method() function called.

**public void method(){**

**int** off =0;//fragmentation offset

**int** flag = 0;//fragmentation flag

message.insert(0,*seqno*);

String sp = **null**;

sp=message.toString();

**while**(**true**){

**try**{

**byte**[] buffer = **new** **byte**[128];

InetAddress address = InetAddress.*getByName*("localhost");

buffer = sp.getBytes();

System.***out***.println(buffer.length);

p = **new** DatagramPacket(buffer, buffer.length, address, port);

s.send(p);

buffer = **new** **byte**[128];

p = **new** DatagramPacket(buffer, buffer.length);

s.receive(p);

*seqno*++;

String received = **new** String(p.getData(), 0, p.getLength());

System.***out***.println(received);

port = p.getPort();//port no of the server socket to which the next  
 packets should be sent to from the client

**if**(flag == 0){

**break**;}}

1. requestFile() method(P2P Client implementation)

When the user gives 3 as the input, a call is made to this function where a client socket is created for communication with the P2P sever it wishes to request a file from.

**public** **void** requestFile(String request){

Socket client = **null**;

PrintWriter out1 = **null**;

**try**{

String rq = "GET "+request.split(" ")[0]+" HTTP/1.1\r\n";

client = **new** Socket(InetAddress.*getByName*(request.split(" ")[1].split("/")[1]),6787);

System.***out***.println(client.getLocalPort());

OutputStream outToServer = client.getOutputStream();

outToServer.write(rq.getBytes());

InputStream inFromServer = client.getInputStream();

BufferedReader in = **new** BufferedReader(**new** InputStreamReader(inFromServer));

...

...

}**catch**(IOException e)

{

e.printStackTrace();

}}

* **Listen.java** (P2P Sever Implementation):

**public** **class** Listen **implements** Runnable{

**private** **boolean** running = **true**;

**private** ServerSocket SSocket = **null**;

Listen() **throws** IOException{

SSocket = **new** ServerSocket(6787,10);

}

**public** **void** run(){

**while**(running){

**try**{

Socket p2p = SSocket.accept();

System.***out***.println(p2p.getInetAddress().toString());

Thread t = **new** Thread(**new** PeerConnection(p2p));

t.start();

}**catch**(IOException e){

//System.out.println("No peer connections");

}}}

**public** **void** kill(){

running=**false**;

}}

* **PeerConnection.java**(Thread created in Listen.java for every file request from a P2P client)

**public** PeerConnection(Socket p2p) **throws** IOException

{

**this**.p2p = p2p;

}

**public** **void** run()

{

**int** statusCode = 0;

String statusMessage = **null**;

String response = **null**;

**int** flag = 1;

**try**

{

BufferedReader inFromClient = **new** BufferedReader(**new** InputStreamReader(p2p.getInputStream()));

String clientSentence=inFromClient.readLine();

String[] request = clientSentence.split(" ");

OutputStream os = p2p.getOutputStream();

...

...

**byte** [] mybytearray = **new** **byte** [(**int**)myFile.length()];

FileInputStream fis = **new** FileInputStream(myFile);

BufferedInputStream bis = **new** BufferedInputStream(fis);

bis.read(mybytearray,0,mybytearray.length);

os = p2p.getOutputStream();

os.write(response.getBytes());

os.write(mybytearray,0,mybytearray.length);

System.***out***.println("Sending...");

os.flush();

bis.close();

p2p.close();

}**catch**(IOException e){

e.printStackTrace();

}}

**Status codes 200, 400, 404, 505 implemented in this class.**

Ex: For status code 404

File myFile = **new** File(request[1]);

**if**(!myFile.exists()){

statusCode = 404;

statusMessage = "Not Found";

flag =0;}

* **UDP Server (ServerUDP.java):**

1. Main method invoked on instantiation.

**public** **static** **void** main(String[] args) **throws** IOException {

ServerUDP DS = **new** ServerUDP();

DS.start();

}

}

1. run() method

**while**(**true**){

**try**{

String name= "Client"+*i*;

**byte**[] buffer = **new** **byte**[2048];

p = **new** DatagramPacket(buffer, buffer.length);

s.setSoTimeout(5000);//timeout

s.receive(p);

InetAddress address = p.getAddress();

**int** cp = p.getPort();

System.***out***.println(cp);

**if**(*i*==1 || !(cp == portno)){

DatagramSocket s1 = **new** DatagramSocke  
 t = **new** Thread(**new** ClientConnection(s1,p,name,address,cp));

t.setName(name);

t.start();

*i*++;

...

...

}**catch**(SocketTimeoutException e){

}**catch**(IOException e){

e.printStackTrace();

}

}

A new thread is created for every new client connection.

* **ClientConnection.java**

**run() method:**

**while**(**true**){

**try**

{

**if**(received == **null**){

**try**{

buffer = **new** **byte**[128];

packet = **new** DatagramPacket(buffer, buffer.length);

socket.receive(packet);

}**catch**(SocketTimeoutException e){

**continue**;

}

}

received = **new** String(packet.getData(), 0, packet.getLength());

...

...

}

}

Call is made to the respective function depending on the request message received from the client.

**public synchronized void informAndUpdate(){**

**while**(**true**){

**try**{

**//fragmentation logic**

**...**

**...**

FileWriter file = **new** FileWriter("list\_server.txt", **true**);

out1 = **new** PrintWriter(file);

**byte**[] buffer = **new** **byte**[2048];

packet = **new** DatagramPacket(buffer, buffer.length);

String[] content = received.split("\r\n");

**int** count = j;

**while**(j < content.length){

System.***out***.println("inside");

out1.println(content[j]+"/"+ClientName);

System.***out***.println(content[1]);

j++;

}

...

...  
}

**finally** {

System.***out***.println("Inside finally");

out1.close();

}

}

}

**public** **void** queryForContent(InetAddress address){

**while**(**true**){

**try**{

**while**(Line != **null**){

**if**(Line.contains(filename)|| filename.equals("all")){

message.append(Line+"\r\n");

}

Line = r.readLine();

}

message.append("\r\n");

...

...

//fragmentation logic

}

}}

**public** **void** exit(){

**try**{

File inputFile = **new** File("list\_server.txt");

File tempFile = **new** File("myTempFile.txt");

BufferedReader reader = **new** BufferedReader(**new** FileReader(inputFile));

BufferedWriter writer = **new** BufferedWriter(**new** FileWriter(tempFile, **true**));

String currentLine;

**while**((currentLine = reader.readLine()) != **null**) {

**if**(currentLine.contains(received.split("/")[1])){

**continue**;

}

**else**{

writer.append(currentLine+"\n");

}

}

writer.close();

reader.close();

**boolean** successful = inputFile.delete();

System.***out***.println("Delete successful"+successful);

successful = tempFile.renameTo(inputFile);

System.***out***.println(ClientName+"Exit"+successful);

...

...

}

**catch**(IOException e){

e.printStackTrace();

}}}

**DATA STRUCTURES, DESIGN CHOICES**

DATA STRUCTURES:

A Data Structure is a set of types, a designated type from that type set, a set of functions, and a set of axioms. This definition implies that a Data Structure is a type with implementation. In our object-oriented programming era, *type with implementation* means class.

The Data Structures used in our P2P File Sharing Project are:

* **StringBuffer:** The StringBuffer class represents a variable-length sequence of characters. StringBuffer objects are used in computations that involve creating new String objects. The StringBuffer class provides various methods for working with StringBuffer objects, including insert(), append() and delete() methods that add/removes characters to a StringBuffer and methods that fetch the contents of StringBuffer objects.
* **Class:** A class is also a Data Structure which contains various data fields and methods that operate on data fields. The classes that we have used in our project are ServerUDP, Client\_UDP, Listen, ClientConnection, and PeerConnection.
* **Files:** Files is also a Data Structure. We are using Files both at the Client-End and the Server-End. Files at Client-End are used to Inform and Update the Directory Server and those at Server-End are used to maintain the information and answer the Query of the Client.

DESIGN CHOICES:

1. Allowing multiple updates to the directory file at the directory server simultaneously when multiple clients issue an “Inform and Update”.

**Implementation**: Synchronization between threads

-Using keyword “synchronized” such that only one thread is allowed at a time to write in the file and others go in wait state. When the current thread is done, it issues a notify() command such that any one of the waiting threads can write to the file next.

1. Handling multiple client requests at the centralized directory server by listening at a fixed port, creating a new socket each time a new client request is to be serviced and handling the requests of different clients simultaneously.

**Implementation**: Multithreading

1. Peer-to-peer transfer involving multiple requests from P2P clients to a single transient P2P server and maintaining the flow of control logic in the main thread running for Client UDP such that it listens on a TCP socket for file requests also continuously and simultaneously presents a user interface to interact with the directory server using UDP.

**Implementation**: Multithreading

-When the client program is run, two threads are created. One thread has the server TCP   
 socket continuously listening on a fixed port and the second one as the main thread which  
 handles communication with the centralized directory server.

1. The server should ignore retransmitted packets sent by the client on not receiving an acknowledgment by keeping a tab on the sequence number of the packets received, but should acknowledge the same packet again so that the client gets to know that it has received the packet.

1. When the client issues an “Exit” command, three threads should be closed according to the implementation in our project code after deleting the entries of the client from the centralized directory server.

**Implementation**: Killing the thread which continuously listens on a TCP socket at the client UDP, deleting the entries corresponding to this client from the directory file, and breaking out from the run() methods of the main thread running at the client UDP and the corresponding thread created at the directory server for this client.

1. Fragmentation of messages greater than the maximum transmission unit of 128 bytes and keeping a tab on when the last fragment is sent.

**Implementation**:

Checking for the size of the message if it is greater than 128 bytes and fragmenting it accordingly before sending it. The receiver or sender can know whether it is the last fragment of the message or an intermediate one by checking the value of the flag bit which is one if it is intermediate and zero if it is the last fragment.

**USER GUIDE**

The code is developed using

* Java on Eclipse v4.4.0
* jdk(Java Development Kit)v1.8.0\_60
* jre(Java Run Environment) 1.8.0\_60

P2P File Sharing Application:

1. Create a workspace (asked first time when it opens after installation) in Eclipse where all the codes you write will be saved.

2. Create a project and load all the files into it.

3. Compile the entire project. Right click on the project folder shown at the left hand side and choose 'Build Project'.

4. Run ServerUDP.java

5. Run ClientUDP.java

6. A user interface will be shown. Give an input for any request to the SeverUDP.java (Three requests - Inform and Update, Query for Content, Exit)

7. Input 1(Inform and Update) - will update the list of files you have in the file at Server.

8. Input 2(Query for Content) After that either enter as requested or enter 'all' if you wish to retrieve the entire list of files at the directory.

9. Input 4 will enable you to request a specific file from any of the hosts(list of hosts having that file displayed as a result of the request: 'Query for Content'). Enter the filename and hostname as asked and displayed in the console. If successful a new file can be seen created at the workspace which is the one you requested or otherwise an appropriate status code and status message will be sent by the host to which you have requested the file.

10. The same can be if two or more clients or peers are to be involved with each client running on a different computer and the sever running on another.(Multithreading has been implemented to avoid erroneous interaction when many clients are involved in the communication)

11. If you wish to quit, issue an exit command (Input 3) which closes the client automatically and the server needs to be terminated manually.

**REFERENCES**

1. <http://docs.oracle.com/javase/7/docs/api/allclasses-noframe.html>
2. <http://www.tutorialspoint.com/java/java_networking.htm>
3. [https://docs.oracle.com/javase/tutorial/networking/datagrams/definition.html](http://www.tutorialspoint.com/java/java_networking.htm)
4. [http://www.tutorialspoint.com/java/java\_multithreading.htm](http://www.tutorialspoint.com/java/java_networking.htm)