ASSIGNMENT 2 REPORT

EECS 3214

Abstract

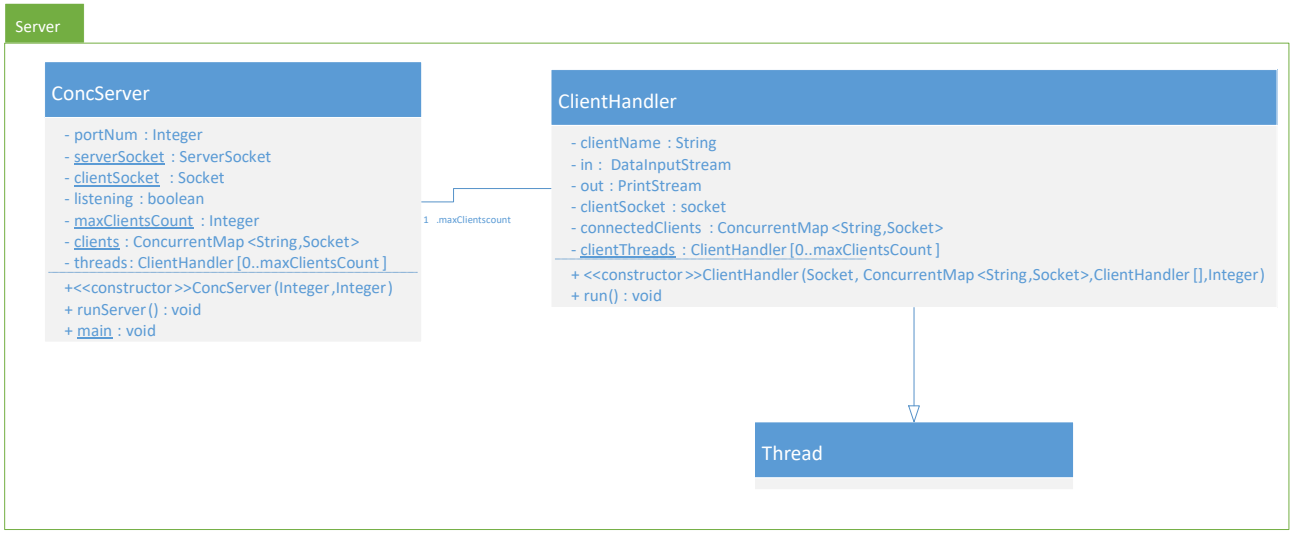
This is Project Report for Assignment 1. It contains Design Decisions, Program structure, Compile and Run information. It also contains list of issues and possible improvements.

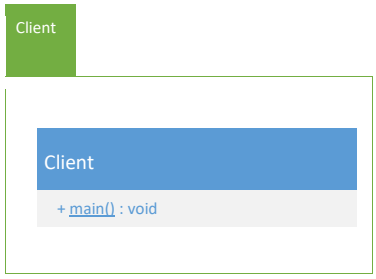
Amanpreet Walia

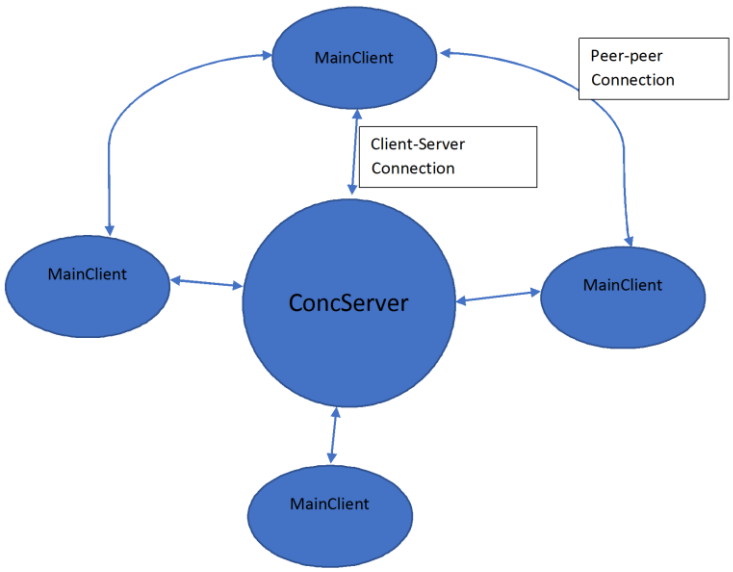
212938734

Design of Server and Client

UML Diagram for Client and Server Class







Peer-Peer Architecture of Messaging System

Design Decisions and Working Client-Server Architecture

For Client/Server model to work, Server side must be running beforehand otherwise we will get exception while connection from Client side.

Server program begins creating a new ServerSocket for server with given port number as shown:

```
 try
  {       serverSocket = new
ServerSocket(portNum);
 } catch (IOException e)
{
     System.out.println(e.getMessage());
    System.exit(-1);
 }
```

After server program, has been started, it waits for an incoming connection inside an infinite while loop like this:

```
 while (listening) {
 try {
        //Accept connection from client socket
 clientSocket = serverSocket.accept();
```

Call to accept method is blocking which keeps waiting until a request from client side has arrived. After a request, has arrived, a new socket is created which establish connection between server and client.

A Client will connect to this server by creating a socket with hostname of server machine and port number of the server process.

```
 //Client Socket to connect to server
Socket echoSocket = new Socket(hostName, portNumber);
```

Once socket has been created on server and client side for connection, this socket will be passed on to a Client Handler thread on server side which deals with all the request of client on behalf of server inside its run method. Clearly there might be some thread getting free inside process , therefor for synchronization this loop is put inside synchronized block.

```
 int i = 0;
  synchronized (this){
 for (i = 0; i < maxClientsCount; i++) {
 if (threads[i] == null) {
              (threads[i] = new
 ClientHandler(clientSocket,clients,threads,maxClientsCount)).start();
break;
          }
      }
```

The thread object is passed with other information like data structure holding name and address of clients, all the other thread objects, and maximum number of threads available for process.

Since we can only connect limited number of clients, all additional clients are notified with server being saturated and their connection will be closed.

```
if (i == maxClientsCount) {
     PrintStream os = new PrintStream(clientSocket.getOutputStream());
os.println("Server too busy.Try later.");
     os.close();
clientSocket.close();
```

In this way, main server program can accept connection from multiple clients while their request will be handled by separate thread objects. This leads to **concurrent behavior of server**.

Once the connection has been established, Input and Output stream must be created on both client and server side to send and receive messages.

Creating Input/output Stream

On Client side, we can use Buffer Reader class to create an input stream to receive response from server. To send message to server, we have use Print Writer class to send queries to server.

```
 try{
       // Input and Output Streams for reading and writing to server
       PrintWriter out =   new PrintWriter(echoSocket.getOutputStream(), true);
       BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
 } catch (IOException e2) {
     System.err.println("Couldn't get I/O for the connection to " + hostName);
```

This is done inside try catch block to catch any exceptions.

On the server side, the DataInputStream is used to receive inputs from the client.The class DataInputStream allows you to read lines of text and Java primitive data types in a portable way. It has several read methods such as read, readChar, readInt, readDouble, and readLine.

Similarly class Printstream is used to send data to client side.It uses print method to send data to client side.

```
 DataInputStream in;
PrintStream out;  try
{
     /*
 * Create input and output streams for this client.
 */     in = new
 DataInputStream(clientSocket.getInputStream());      out =
 new PrintStream(clientSocket.getOutputStream());
 } catch (IOException e) {
System.out.println(e);
 }
```

To store all the connections, Concurrent HashMap is used. It stores name, client socket for each client.

```
//Data Structure to keep track of all the connected clients private
 ConcurrentMap<String,Socket> clients = null;


 clients = new ConcurrentHashMap<String, Socket>();
```

This is data structure is passed to each thread by reference so that each thread can get information of all the clients connected to system. Instead of using normal Hash Map, ConcurrentHashMap is used to allow synchronized access to this data structure from various threads.

Once a client asks to join the Server using JOIN command, it is being asked to provide a unique name to identify itself. This name along with the respective socket is then added to the data structure. This step involves various procedures of defensive programming to avoid conflicting data.

```
if (input.matches("JOIN")) {
    if(this.clientName == null){
        out.println("Enter a unique name to identify yourself");
        name = in.readLine();
        while((name == null)||(name.isEmpty())) {
            out.println("Name cannot be null or Empty!");
name = in.readLine();
        }         while(connectedClients.containsKey(name)||(name ==
null)||(name.isEmpty())){
K             wkdlkbwbdbwbdlwkbdlwbldbwlbldblwbldbwif((name == null)||(name.isEmpty  ())){
                out.println("Name cannot be null or Empty!,Enter a unique name");
            }
else
                out.println("Name exists in database,Enter a unique name");
            name = in.readLine();
        }
        //Add the client socket and client name to concurrent map
connectedClients.put(name,clientSocket);
        this.clientName = name;
        //Check if client has been successfully added or not
if (connectedClients.containsKey(name))
            out.println("Client Successfully connected to server by name:
"+this.clientName);
else
            out.println("Unable to connect client to server due to unknownd!!");
}
```

```
else
    out.println("You are already connected!!");
```

When a client wants to list all the other users connected to server, it sends LIST command to server which is processed by concatenating list of all the connected users to a string and sending it as a response to client.

```
if  (input.matches("LIST")) {

    if(this.clientName != null){

        if (connectedClients.containsKey(this.clientName)) {
            StringBuilder response = new StringBuilder("List of Clients Connected: ");
      int num = 1;
            //append IP address of clients to this string
      for (String names : connectedClients.keySet()) {
response.append(" "+num+"."+names+
"(" +connectedClients.get(names).getInetAddress().toString().substring(1)+")");
num++;
            }

            out.println(response.toString());
        }
      }
else
            out.println("Unable to list connected clients, You have to JOIN
first"); }
```

When a client wants to Leave the server, it sends LEAVE to server which is processed by removing it from Hash Map, then closing the sockets and data streams followed by freeing the thread for use by other clients.

```
else if (input.matches("LEAVE")) {
      //close the socket for this client and put thread to
null     if(this.clientName != null){          if
(connectedClients.containsKey(this.clientName)) {

            connectedClients.remove(this.clientName);
break;// break from while loop
        }
    }        else {          out.println("You
have to join first");           }

}
try {
      //Make this thread empty to be used by other
clients     synchronized (this) {              for (int
i = 0; i < maxClientsCount; i++) {              if
(clientThreads[i] == this) {
      clientThreads[i] = null;
            }
        }
    }
      //Close input and output stream and close
socket     this.out.close();
      this.in.close();
this.clientSocket.close();
}
catch (IOException ex) {
    System.out.println("Error closing the socket and streams");
}
```

When a client is disconnected from server, its data streams will only be returning null. This is used to determine on client side that connection has been terminated and resources for sockets and data streams should be freed now.

```
if(data!= null) {
     System.out.println(data);
    System.out.print(">> ");

} else{
    System.out.println("Disconnected from server");
     //Closing the streams and sockets
 in.close();
    out.close();

echoSocket.close();
System.exit(1);  }
```

Since we must enquire Central server about the respective peers connected, to obtain the information about peer by nickname, another query is added. This query is used to obtain host IP address and local server port by its nickname published on Central server list.

```
else if (input.matches("P2P CONNECT .*")) {

    if (this.clientName != null) {

        if (connectedClients.containsKey(this.clientName)) {
            peerName = input.split(" ")[2];
            if (connectedClients.containsKey(peerName)) {
                if (this.clientName.matches(peerName)) {
                    out.println("ERROR: Illegal connecting to yourself!!");
                } else {

                    StringBuilder peerInfo = new StringBuilder();
                    peerInfo.append("FOUND: ");

peerInfo.append(connectedClients.get(peerName).getFirst().getInetAddress().toString()
.substring(1));
                    peerInfo.append(",");
                    peerInfo.append(connectedClients.get(peerName).getSecond());
                    out.println(peerInfo);
                }

            } else {
                out.println("ERROR: No such peer exists!!");
            }
        }
    } else
        out.println("Unable to establish P2P connetion to clients, You have to JOIN
first");
}
```

P2P Architecture and Design Details

Client Class

This class provides basic functionality of a client. It provides abstract methods like `writeToPeer` and `readFromPeer` for communication with server. This class is also responsible for closing the socket connection.This class is as parent class for all client related functionality.

```
public String readFromPeer() {
    try {
        return in.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }

    return null;

}

// Write data to server side
public void writeToPeer(String data) {
    out.println(data);
    flush();

}

public void flush() {
    out.flush();
}

public void closePeerConnection(){
    try{
        //this.in.close();
        //this.out.close();
        this.clientSocket.close();

    }catch(Exception e){
        System.out.println(e.getLocalizedMessage());


    }

}
```

PeerClient Class

This class inherit the functionality of Client plus implements threading to have an independent messaging system from main program. Since the messages are received in threads, multiple clients can print messages on output.

```
public PeerClient(String hostName, int port,PeerNode mainNode){
    super(hostName,port);
    ref = mainNode;
    //messages = new LinkedList<String>();
}
```

```
public void run() {
    while(true){
        String message = this.readFromPeer();
        if(message == null){
            stopClient();
        }
        else if(message.matches("P2P-DISCONNECT-NOW")){

ref.connections.remove(this.clientSocket.getInetAddress().toString().substring(1));
        }
        else{
            System.out.println(host+":"+message);
        }
    }
}

public void stopClient(){
    t.stop();
    closePeerConnection();
}
```

PeerClientHandler and PeerServer

These classes are used to provide Server functionality to the node. Their functionality is exactly same as defined for Client-Server architecture except the functionality is reduced to only act as message reader and writer. These Classes provide the functionality of connecting to remote client and share messages.

PeerNode Class

This is the most important class which provide each abstract functionality of acting as client as well as server together. This class acts as additional interface to the main client connected to central server which provides p2p messaging ability. This class contains storage for each connected peer and hence provide functionality of adding and removing peers. It also provides functionality of basic mesaaging with hostname.Since any peer connected can be client as well server, this class provides abstraction of automatically deciding which socket to choose to deliver the message to the corresponding peer.

```
/*
 * This method is used to connect this peer with other peers, a TCP connection is setup between this
peerNode(acting as Client)
 * and other as Server, if a connection exists from other side already , this will not do anything
 */
public void addConnection(String host, int port) {
    if(!conn_clients.containsKey(host)&&(!connections.containsKey(host))){
        PeerClient client = new PeerClient(host, port,this);
        client.t = new Thread(client);
        client.t.start();
        conn clients.put(host,client);
        System.out.println("Successfully connected with client: "+ host+"on port: "+port);
    }
    else{
        System.out.println("Error connection already exists, Illegal reconnecting!!");
    }
}
```

```
public void sendMessage(String hostName, String message) {
    //This peer is connected as Server to the corresponding peer
    if(conn_clients.containsKey(hostName)){
        PeerClient toMesClient = conn_clients.get(hostName);
        toMesClient.writeToPeer(message);
    }
    //This peer is connected as Client to the corresponding peer
    else if(connections.containsKey(hostName)){
        // Do the server sending stuff here
        server.getHandler(hostName).sendMessage(message);
    }
    else{
        System.out.println("Error: No such connection exists with the hostname");
    }
}

public void closeConnection(String hostName){
    if(conn_clients.containsKey(hostName)){
        PeerClient cl = conn_clients.remove(hostName);
        cl.writeToPeer("P2P-DISCONNECT-NOW");
        cl.stopClient();
    }
    //This peer is connected as Client to the corresponding peer
    else if(connections.containsKey(hostName)){
        server.getHandler(hostName).closeConnection();
    }
    else{
        System.out.println("Error: No such connection exists with the hostname");
    }
}

public void closeEveryThing(){
    for(String key :conn_clients.keySet()){
        closeConnection(key);
    }
    for (String key:connections.keySet()){
        closeConnection(key);
    }

}
```

MainClient Class

As apparent from its name it is the main class of whole peer architecture. This class act as main client for client server architecture as well provide P2P capabilities using PeerNode class. This class is amalgam of P2P Architecture as well as Client-Server architecture.

Working

In central server clients are added using Client-Server architecture. For this host address of the central server needs to be known all the nodes wish to participate in network. Since all the nodes act as peers providing same level functionality, nodes which wish to connect to other node retrieve its address and port number from central server. This is followed by opening TCP connection to the other node. After this node interact with each other using the TCP link. If any node wants to close the connection, it can do so by asking PeerNode class to close the link. When a node leaves the central server , it close all its corresponding connections beforehand.

Situations where Program will not work Properly

Abrupt ending of Node process

If the client process get killed due to some reasons (Ctrl + c, closing terminal window), Server program is not able to free its resources for that client. Due to this, other clients who want to connect will not able to connect due to limited number of client threads even if number of clients connected is less than maximum number of clients which server can handle. Since clients which are currently running are still present in database of main server, It conveys false information to the peers.

Knowledge of Busy Server to Clients

If number of clients connected to server is equal to its maximum limit and other clients want to join the server, knowledge of server being busy is relayed to them only when they send JOIN command. Due to this some client processes, will be under false pretense that they can communicate with server. This however doesn't effect normal working of the programs.

Stop Server Process

There is no way to close the server process other than explicitly killing it.

More than one node running with same IP address

An important assumption is a made while designing the distributed messaging system to have only one peer node from a one IP address. This means that IP address can uniquely identify each Node in messaging system. This is assumption is reasonable in context that most modern chat clients have singleton structure which allow only one instance to run on single machine.

Improvements Possible

Given more time for this assignment following improvements are possible:

1. **Catch Explicit Closing of Clients**
   For Client side programming, it can be programmed to catch interrupt signal (Ctrl + c) and send '**LEAVE**' message to server side so that resources from server side can be freed for other clients. On Server side, all client connections can be scanned in loop for whether they are alive or not and same procedure can be applied on server side to free the resources.

2. **Appropriate output from server side**
   Currently data sent from server side must be in single line to be received appropriately by client side. However, this can be improved by reading the input data stream in loop till we have no data left and displaying it in more meaningful way.

3. **Stop Server Process in Appropriate manner**
   When closing server side process, a separate thread can be used to scan input from server side to decide when to close server process. In this way, all connections to clients can be closed in more appropriate manner.

4. **GUI Application for Server and Client**
   Currently client and server have command line interface which can be improved by building Graphical User Interface.

5. Host Nickname Dictionary Search

Currently all the communication with peers is done using their respective IP address. However, this can be improved to use only nicknames for peers as each IP address is uniquely identified by nickname.

Running and Compilation

Compilation

Centralized Server Side

To compile Client program:

1. Move to submission folder and type `cd Server` in Terminal to move to Server Directory.
2. Type `javac ConcServer.java ClientHandler.java MyPair.java` in Terminal to compile the Server program

To run the program, type `java ConcServer <Port Number> <No. of threads for Clients>`

e.g. **`java ConcServer 2124 3`** will start Server on Port Number 2124 on the machine and it will accept maximum 3 clients to join the Server**.**

Peer Side

To compile Client program:

1. Move to submission folder in Terminal and type `cd Peer` to move to Client Directory.
2. Type `javac Client.java MainClient.java PeerClient.java PeerClientHandler.java PeerNode.java PeerServer.java` in Terminal to compile the Client program.

To run the program, type `java MainClient <centralized Server IP Address> <Local Server IP Address>`

e.g. **`java MainClient 130.63.94.21 2124 3295`** will start Peer Node and connect it to centralized server on 130.63.94.21 with Port Number 2124 and start a local Server on port 3295.

Run Commands

Commands

JOIN : Join the centralized Server for Discovery. Followed by asking for a unique name

LIST : List the available peers connected to centralized Server

LEAVE : Disconnect from centralized server as well as all connected peers

P2P CONNECT *nickname*: Connect to peer on centralized server with corresponding nickname assigned.

P2P SEND<HOST IP ADRESS>: *message*: Send message to the respective peer on host IP address, this will fail if peer connection is not established

P2P DISCONNECT <HOST IP ADRESS>: Disconnect from the peer, this will fail if there is no connection to peer.

Sources for Program Skeletons

1. http://makemobiapps.blogspot.ca/p/multiple-client-server-chat-programming.html

2. http://www.eecs.yorku.ca/course_archive/2016-17/W/3214/ASSGMTS/EchoClient.java