Lab07: Building a Chat System
CIS 457
Due by: 10/30/ 2018
Total Points: 20 Points
Submission format: hardcopy report per group of 2 students

Lab Objectives

The purpose of this lab is to:

- Build a multi-threaded chat system using traditional client/server architecture.
- Build a chat system using IP multicasting.

Introduction

Chat systems are not only popular on the Internet but also belong to the typical applications in distributed computer networks. A chat is a textual conversion between several participants communicating with each other. Participants must register for chart service, which is offered by a server, and can then send messages to this chat and follow all messages sent to this chat. Each participant builds up a connection to the server. The server manages the connections and sends incoming messages to all participants. Sockets represent a point to point connection. Therefore, the server has to simulate a multicast by many individual unicasts. We will implement the *server multicast scenario* using two different methods such as follows.

Part I: Multi threaded Chat System (10 Points)

Use the java source files given to you in this lab. In this part, six classes are required for this chat system:

- **ChatClient** on the client side,
- **ChatSever** on the server side,
- **ChatHandler** which undertakes the processing of the message for the server,
- **ChatFrame** class to implement the GUI on the client side,
- **EnterListener** class waits for input in the GUI. It reads the input and sends it to the ChatClient by means of the sendTextToChat() method,
- **ExitListener** listens to the closing of the input window and then calls the disconnect() method to give the client the possibility to unregister properly from this chat.

➔ In the following code, wherever you see "?", you need to supply a missing detail. After completing the code, compile it using: javac -deprecation *.java **Ignore** any compilation warnings.

The following program implements the ChatClient at the client side:
Note that the *ChatClient and ChatServer* implement the I/O streams using the *DataInputStream and DataOutputStream without buffering* because client and server send and receive a small amount of character data that doesn't worth to buffer it. However, the *ChatHandler* waits for data from clients, buffer it then sends it to the handler of each client. The *ChatHandler* uses *buffered I/O streams*.

// ChatClient.java

```
import java.net.*;
import java.io.*;
import java.awt.event.*;

public class ChatClient {

  public ChatFrame gui;

  private Socket socket;
  private DataInputStream in;
  private DataOutputStream out;

  public ChatClient(String name, String server, int port) {
    // GUI Create GUI and handle events:
    // After text input, sendTextToChat() is called,
    // When closing the window, disconnect() is called.

    gui = new ChatFrame("Chat with Sockets");
    gui.input.addKeyListener (new EnterListener(this,gui));
    gui.addWindowListener(new ExitListener(this));

    // (1) create a socket, (2) register and (3) listen to the server

    // 1. create a socket
    try {
      socket = new Socket(server, port);
      in = new DataInputStream(?);
      out = new DataOutputStream(?);

    // 2. register to the server
      out.writeUTF(name);

    // 3. listen to the server
      while (true) {
        gui.output.append("\n"+in.readUTF());
      };
    }     catch (Exception e)        {
      e.printStackTrace();
    }
  }


  protected void sendTextToChat(String str) {
    try {
// Create and write str to the chat server via the output stream
      out.?;
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  protected void disconnect() {
    try {
//Close the socket
      socket.?;
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  public static void main (String args[])throws IOException {
    if (args.length!=3)
      throw new RuntimeException ("Syntax: java ChatClient <name> <serverhost> <port>");
    int port=Integer.parseInt(args[2]);
    ChatClient c=new ChatClient(args[0], args[1], port);
  }
}
```

On the server side, the program is divided into *two parts*: (1) the ChatServer which accepts the calls and (2) the ChatHandler thread which is passed a single call by the ChatServer and processes it. The ChatServer waits for new incoming connections, reads the name of the new client via its welcoming socket and then starts the ChatHandler, which takes care of the connection, so that the server can again reply quickly to other connection requests. The following program implements the ChatServer program:

//ChatServer.java

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ChatServer {

  public ChatServer (int port) throws IOException {

ServerSocket server = new ServerSocket (port);

    while (true) {
      Socket client = server.accept();
      DataInputStream in = new DataInputStream(?);
      String name = in.readUTF();
//return the client's IP address
      System.out.println ("New client "+name+" from " + ?);
      ChatHandler c = new ChatHandler (name, client);
//Start the thread     ?
    }
  }

  public static void main (String args[]) throws IOException {
    if (args.length != 1)
      throw new RuntimeException ("Syntax: java ChatServer <port>");
    new ChatServer (Integer.parseInt (args[0]));
  }
}
```

Now, the *ChatHandler* is implemented as a thread, so every new handler has its own control flow and they can be executed concurrently. At creation time, the handler initializes its socket connection with the client for which it is responsible. Together, the *handlers* manage a *global list (vector)*, in which each of them registers, thus being able to reference all others. This is required to simulate a broadcast. The handler waits for incoming contributions of the client by constantly trying to read from the InputStream. When it receives a message, it sends it to the OutputStream of every handler in the global list.

The following program implements the ChadHandler:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ChatHandler extends Thread {

  Socket socket;
  DataInputStream in;
  DataOutputStream out;
  String name;

  protected static Vector handlers = new Vector ();

  public ChatHandler (String name, Socket socket) throws IOException {

    this.name = name;
    this.socket = socket;
    in = new DataInputStream (?);
    out = new DataOutputStream (?);

  }



  public void run () {

    try {
      broadcast(name+" entered");

// add the client handler to the handlers’ list
      handlers.addElement (this);

// wait for client contributions by constantly trying to read from the InputStream
      while (true) {
        String message = ? ;
//Broadcast the user’s name and the message
        ?

      }

    } catch (IOException ex) {
      System.out.println("-- Connection to user lost.");
    }

finally {
      handlers.removeElement (this);

//Broadcast that user left
      ?
      try {

//close the socket
       ?

      } catch (IOException ex) {
        System.out.println("-- Socket to user already closed ?");
      }
    }
  }

  protected static void broadcast (String message) {
    synchronized (handlers) {

// For more info about the Interface Enumeration, please consult the java documentations.

      Enumeration e = handlers.elements ();

  // Test if e enumeration has-more-elements
         while (e.?)) {

// The first time you call nextElement(), it returns the element at position zero. In subsequent method calls, nextElement()
returns the element at position one, two, three, and so on.
        ChatHandler handler = (ChatHandler) e.?;

 try {
```

```
// sends the message to the OutputStream of every handler in the global list using the writeUTF method and flush this
OutputStream
          handler.out.writeUTF (?);
                 ?

        } catch (IOException ex) {
          handler.stop ();
        }
      }
    }
  }
}
```

Question 1

a) Compile and run the ChatServer using "java ChatServer 1234", and run two clients using "java ChatClient ss1 localhost 1234 AND java ChatClient ss2 localhost 1234 ". Test the application and ensure it works by entering a hello message in the input box from each chat window. Then, provides a screen capture for the testing process as well as the programs code.

b) What is the purpose of using the "synchronized" keyword in the implementation of the broadcast() method?

Part II: Building Chat System using IP Multicast (10 Points)

This part demonstrates the implementation of a chat system communication using IP multicast. Java provides multicast socket in the *java.net* package, which can be used to communicate with a group of hosts. The Java multicast support is available at the Internet protocol level (IP multicasting). IP multicasting uses the IP range *224.0.0.1 to 239.255.255.255*. The range of IP 244.0.0.1 to 224.255.255.255 is reserved for multicast routing information. All other addressees can be freely used for group communication. All participants in such a communication group must first register with the group to be able to join the communication. To end the membership in the group, simply the participant must sign out. *IP multicast uses UDP transport protocol* rather than TCP. Note that the sender does not necessarily have to be registered for a group.

As opposed to socket communication, the implementation of a chat system with multicast no longer requires a server to receive and forward the messages, but to send them directly to anybody listening on the group's multicast IP address. Thus only one core class, ChatClient, is required beside the GUI classes.

- ➔ Wherever you see "?", you will need to supply a missing detail (s) then compile the code.

// ChatClient.java

```
import java.net.*;
import java.io.*;

public class ChatClient {

  ChatFrame gui;
  String name;

  InetAddress group;
  MulticastSocket socket;
  int port = 6789;

  public ChatClient(String name) {
    this.name = name;

    // GUI Create GUI and handle events:
    // After text input, sendTextToChat() is called,
    // When closing the window, disconnect() is called.

    gui = new ChatFrame("Chat with IP-Multicast");
    gui.input.addKeyListener (new EnterListener(this,gui));
    gui.addWindowListener(new ExitListener(this));


// Step 1:
// In order for a host to receive a multicast data,
// the receiver must register with a group, by creating a MulticastSocket on a port and call the joinGroup() method.

    try {
      socket = new MulticastSocket(port);
      group = InetAddress.getByName("226.1.3.5");
      socket.joinGroup(group);
      gui.output.append("Connected...\n");


// Step 2:
 // waiting for and receiving messages

      while (true) {
            byte[] buffer = new byte[1000];

// create a place holder for the DatagramPacket
            DatagramPacket datagram = new DatagramPacket(?);
// invoke the socket’s receive () method
            socket.?;
            String message = new String(datagram.getData());
            gui.output.append(message);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

//Step 3:

  public void sendTextToChat(String message) {
    message = name+": "+message+"\n";

// convert the message into bytes
    byte[] buf = (message).?;

// construct the DatagramPacket
    DatagramPacket dg = new DatagramPacket(?);
    try {

//Send the DatagramPacket
      socket.?;
    }
    catch (IOException ex) {
      System.out.println(ex);
    }
  }

  public void disconnect() {}

  public static void main(String args[]) {
    if (args.length!=1)
      throw new RuntimeException ("Syntax: java ChatClient <name>");
    ChatClient client = new ChatClient(args[0]);
  }
}
```

Question 2

a) Test this part of the lab by running two clients using “java ChatClient ss1 AND java ChatClient ss2”, then provide a screen capture for the output as well as the program code.

b) What are the advantages and disadvantages of using IP multicasting?

c) Modify the code such as the client can send a string Bye to close the chat session. Attach your modified code to the lab report.