Week 9 Lab - A Simple Chat System

COSC244

1 Assessment

This lab is worth 0.5%. The marks are awarded for the written answers to the preparation questions (worth 0.25%) and completing two programming exercises (worth 0.25%).

The preparation questions should be answered **before** coming to the lab. They should be stored electronically in a plain text file with a .txt suffix in your ∼/244/09 directory. They will be checked by the demonstrators at the start of the lab.

Make sure you get your work signed off by a demonstrator before leaving the lab.

2 Handling Multiple Clients

In the last lab, you created *MultiServer.java* which allowed multiple clients to connect to the server. Anything typed into the server was sent to all clients. A line typed on any client was sent to the server. Let's examine *MultiServer's* code which is shown below to see how it operates.

```java
// MultiServer.java

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

public class MultiServer extends Thread {

   private static List<PrintWriter> clients = new LinkedList<PrintWriter>();

   public static void main(String[] args) {
      try {
         int port = Integer.parseInt(args[0]);
         ServerSocket serverSocket = new ServerSocket(port);
         new MultiServer().start();
         System.err.println("Waiting for a client to connect");
         while (true) {
            Socket socket = serverSocket.accept();
            synchronized(clients) {
               clients.add(new PrintWriter(socket.getOutputStream(), true));
            }
            System.err.println("Accepted connection on port " + port);
            new ReadWriteThread(socket.getInputStream(), System.out).start();
         }
      } catch (Exception e) {
         e.printStackTrace();
         System.err.println("\nUsage: java MultiServer <port>");
28         }
29     }
30
31     public void run() {
32        Scanner stdin = new Scanner(System.in);
33        while (stdin.hasNextLine()) {
34           String line = stdin.nextLine();
35           synchronized(clients) {
36              for (PrintWriter client : clients) {
37                 client.println(line);
38              }
39           }
40        }
41     }
42
43  }
```

Since we can have multiple clients, we need some way to keep track of them. This is provided by line 9 which creates a *LinkedList* of *PrintWriters* called `clients`.

Lines 13 & 14 are the same as in *Server.java*. Line 15 starts an instance of *MultiServer*. This results in a call to the `run()` method which is given in lines 31 - 41.

Line 32 assigns `stdin` to a new *Scanner* for *System.in*. Lines 33 - 40 form a `while` loop which continues as long as there is input to read from the keyboard. Line 34 reads the line typed into the *String* variable `line`.

Line 35 uses the `synchronized` keyword. This may be the first time you have seen this keyword. It is applied to `clients` which is our list of clients. What this does is give exclusive access to `clients`. Until the code in lines 36 - 38 completes, no other thread may access `clients`. If we did not do this, it is possible for another thread to modify `clients` in some unpredictable way and chaos would result.

Lines 36 & 37 are a `for` loop on `clients` which iterates through the list of clients. For each client, we print the line which the *Scanner* read. This results in `line` being sent to each client via their own socket.

Let's return to `main()`. Line 16 prints a line so we know something happened when we started the server. Lines 17 - 24 form an infinite `while` loop. Line 18 calls the `accept()` method and waits for a client to connect. The `accept()` method returns a *Socket* which is assigned to `socket`. When a client connects, lines 19 - 21 grant exclusive access to `clients`. Line 20 gets an *OutputStream* on the socket, wraps a *PrintWriter* around it and adds it to the clients list.

Line 22 prints a message on the server screen about accepting a connection. Line 23 creates a new *ReadWriteThread* using the `socket` for input and *System.out* for output, and starts the thread.

In summary, starting the server creates a thread which reads from the keyboard and prints to all connected clients. When a client connects, a thread is started reading from the client's socket and printing to the screen. There is one *ReadWriteThread* for each connected client.

Q1 Explain in your own words why we must use the `synchronized()` keyword when we access `clients`.

Q2 What line of code starts the thread which reads from the keyboard and prints to all clients?

Q3 What lines of code does this thread execute?

3 A Simple Chat System

Client.java along with *ReadWriteThread.java* is simple (and short) and is flexible at the same time. The client prints to a socket whatever it reads from the keyboard and prints to the screen whatever it reads from a socket.

We can use the client in several ways, some of which you saw in the last lab. Now let's implement a simple chat system. We can use *Client.java* and *ReadWriteThread.java* without modification. What we need is a different server.

In our chat system, whatever is typed on the keyboard of any client is sent to all clients (including the one that sent it). We will not type on the server's keyboard. Instead the server receives a line from a client and sends it to all clients. It outputs onto its screen information as clients connect and disconnect, and status information about what it is sending and to whom.

The code for our chat server is given below.

```
 1 // ChatServer.java
 2
 3 import java.io.*;
 4 import java.net.*;
 5 import java.util.*;
 6
 7 public class ChatServer {
 8
 9    private static List<ClientHandler> clients = new LinkedList<ClientHandler>();
10
11    public static void main(String[] args) {
12       try {
13          new ChatServer().startServer(Integer.parseInt(args[0]));
14       } catch (Exception e) {
15          e.printStackTrace();
16          System.err.println("\nUsage: java ChatServer <port>");
17       }
18    }
19
20    public void startServer(int port) throws Exception {
21       ServerSocket serverSocket = new ServerSocket(port);
22       System.err.println("ChatServer started");
23       while (true) {
24          ClientHandler ch = new ClientHandler(serverSocket.accept());
25          System.err.println("Accepted connection from " + ch);
26          synchronized (clients) {
27             clients.add(ch);
28          }
29          ch.start();
30       }
```

```
   }

   public static void sendAll(String line, ClientHandler sender) {
      System.err.println("Sending '" + line + "' to : " + clients);
      synchronized (clients) {
         for (ClientHandler cl : clients) {
            cl.send(sender + ": " + line);
         }
      }
   }

   public static class ClientHandler extends Thread {

      private BufferedReader input;
      private PrintWriter output;
      private String id;
      private static int count = 0;

      public ClientHandler(Socket socket) throws Exception {
         input = new BufferedReader(
               new InputStreamReader(socket.getInputStream()));
         output = new PrintWriter(socket.getOutputStream(), true);
         id = "client_" + ++count;
      }

      public void send(String line) {
         output.println(line);
      }

      public String toString() {
         return id;
      }

      public void run() {
         try {
            send("Welcome! You are " + this + ".");
            String line;
            while ((line = input.readLine()) != null) {
               sendAll(line, this);
            }
         } catch (IOException e) {
            e.printStackTrace();
         } finally {
            synchronized (clients) {
               clients.remove(this);
            }
            System.err.println(this + " closed connection!");
         }
      }
   }
}
```

4 Lab Exercises

4.1 Get the Provided Code

Create and change to a directory you wish to work in for this lab. Type the following to copy files for this lab.

```
cp -r /home/cshome/coursework/244/start/09 .
```

Don't forget the period at the end. Compile *Client.java* and *ReadWriteThread.java* by typing:

```
javac Client.java
```

Client.java and *ReadWriteThread.java* are our solutions to Lab 8.

4.2 *ChatServer.java*

Type the *ChatServer.java* code given above into a file and compile it.

To run these programs and answer the questions below, you will need 4 terminal windows. You might also want to switch to a different workspace on your Linux desktop. Start 4 terminals and move them so they occupy the four corners of your screen. I will refer to them as UL, UR, LL, and LR meaning upper left, upper right, lower left, and lower right, respectively. In each terminal window `cd` to the directory you are working in.

In the LR terminal, start the server by typing: `java ChatServer 7777`

Q4 What is the response when *ChatServer* starts?

In the LL terminal, start a client by typing: `java Client 7777 localhost`

Q5 What is the response of the server when this client is started?

Q6 What is the response in the client window as it starts?

In the UL terminal, start another client by typing the same command.

Q7 What is the response of the server when this client is started?

Q8 What is the response in the client window as it starts?

In the UR terminal, start another client by typing the same command. You now have 3 clients running and connected to the server.

In `client_1`'s window type: `Hello from client 1`

Q9 What is the response in `client_2`'s window?

Q10 What is the response in `client_3`'s window?

Q11 What is the response in the server's window?

Type something into `client_2`'s window and into `client_3`'s window and observe what happens in the other clients' windows and in the server window. Notice that what you type in a particular client appears on its screen as you type. Once you type <*Return*>, what you have typed is sent to each client preceeded by the sending client's ID. This includes the client that sent the message. Notice the status information printed in the server's window.

Type *ctrl-c* in `client_3`'s window to terminate it. You should get a Unix prompt in that window.

Q12 What is the response in the server's window?

Type something in `client_1`'s window.

Q13 What is the response in `client_2`'s window?

Q14 What is the response in the server's window?

Type *ctrl-c* in `client_1`'s window and in `client_2`'s window. Both these programs terminate and you get a Unix prompt. The server outputs messages about each client closing its connection.

In the UL window, start a new client by typing: `java Client 7777 localhost`

Q15 What client are you now?

Terminate this client and the server by typing *ctrl-c* in each window. You will need the answers to these questions to prepare for the week 11 lab.

Make sure you get your work signed off by a demonstrator before leaving the lab.