Multi-user Chat Program

BOFIN BABU

2013h313085H

Documentation

This is the documentation of a multi-user chat program written in Java. The program support command-line chat between multiple users in different PC's with Java installed.

Table of Contents

No	Item
1	Introduction
2	Opening a socket
3	Creating an input stream
4	Creating an output stream
5	Closing a socket
6	Implementation of Client and Server programs
7	Running the chat program

Introduction

A socket is the one end-point of a two-way communication link between two programs running over the network. A network connection is initiated by a client program when it creates a socket for the communication with the server. To create the socket in Java, the client calls the **Socket** constructor and passes the server address and the specific server port number to it. At this stage the server must be started on the machine having the specified address and listening for connections on its specific port number.

```
SERVER                                   CLIENT
    ,--------------.                         ,--------------.
    |            ,-|                         |              |
    |    accept  | |<----------------------|   create     |
    |            `-|                         |              |
    |            ,-|                         |-.            |
    | read/write | | <--------------------->| | read/write |
    |            `-|                         |-'            |
    `--------------'                         `--------------'
```

Opening a socket

The Client side

When programming a client, a socket must be opened like below:

```
Socket MyClient;
 try {
       MyClient = new Socket("MachineName", PortNumber);
     }
     catch (IOException e) {
       System.out.println(e);
     }
```

Where:

- **MachineName** is the machine name to open a connection to and
- **PortNumber** is the port number on which the server to connect to is listening.

The server side

When programming a server, a server socket must be created first, like below:

```
ServerSocket MyService;
 try {
      MyServerice = new ServerSocket(PortNumber);
    }
    catch (IOException e) {
      System.out.println(e);
    }
```

The server socket is dedicated to listen to and accept connections from clients. After accepting a request from a client the server creates a client socket to communicate (to send/receive data) with the client, like below:

```
Socket clientSocket = null;
 try {
      serviceSocket = MyService.accept();
    }
    catch (IOException e) {
      System.out.println(e);
    }
```

Creating an input stream

On the client side, you can use the **DataInputStream** class to create an input stream to receive responses from the server:

```
DataInputStream input;
 try {
    input = new DataInputStream(MyClient.getInputStream());
 }
 catch (IOException e) {
    System.out.println(e);
 }
```

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**. One has to use whichever function depending on the type of data to receive from the server.

On the server side, the **DataInputStream** is used to receive inputs from the client:

```
DataInputStream input;
 try {
     input = new DataInputStream(serviceSocket.getInputStream());
 }
 catch (IOException e) {
    System.out.println(e);
 }
```

Create an output stream

On the client side, an output stream must be created to send the data to the server socket using the class**PrintStream** or **DataOutputStream** of **java.io** package:

```
PrintStream output;
 try {
    output = new PrintStream(MyClient.getOutputStream());
 }
 catch (IOException e) {
    System.out.println(e);
 }
```

The class **PrintStream** implements the methods for displaying Java primitive data types values, like **write** and **println** methods.

On the server side, one can use the class **PrintStream** to send data to the client.

```
PrintStream output;
 try {
    output = new PrintStream(serviceSocket.getOutputStream());
 }
 catch (IOException e) {
    System.out.println(e);
 }
```

Closing Sockets

Closing a socked is like closing a file. You have to close a socket when you do not need it any more. The output and the input streams must be closed as well but before closing the socket.
On the client side you have to close the input and the output streams and the socket like below:

```
try {

    output.close();

    input.close();

    MyClient.close();

 }

 catch (IOException e) {

    System.out.println(e);

 }
```

On the server you have to close the input and output streams and the two sockets as follows:

```
try {
   output.close();
   input.close();
   serviceSocket.close();
   MyService.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

Usually, on the server side you need to close only the client socket after the client gets served. The server socket is kept open as long as the server is running. A new client can connect to the server on the server socket to establish a new connection, that is, a new client socket.

Implementation

The client

The code below is the multi-threaded chat client. It uses two threads: one to read the data from the standard input and to sent it to the server, the other to read the data from the server and to print it on the standard output.

```
import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class MultiThreadChatClient implements Runnable {

  // The client socket
  private static Socket clientSocket = null;
  // The output stream
  private static PrintStream os = null;
  // The input stream
  private static DataInputStream is = null;
  private static BufferedReader inputLine = null;
  private static boolean closed = false;

  public static void main(String[] args) {

    // The default port.
    int portNumber = 2222;
    // The default host.
    String host = "localhost";

    if (args.length < 2) {
      System.out
          .println("Usage: java MultiThreadChatClient <host>
<portNumber>\n"
              + "Now using host=" + host + ", portNumber=" +
portNumber);
    } else {
      host = args[0];
      portNumber = Integer.valueOf(args[1]).intValue();
    }

    /*
     * Open a socket on a given host and port. Open input and output
streams.
     */
    try {
      clientSocket = new Socket(host, portNumber);
      inputLine = new BufferedReader(new InputStreamReader(System.in));
      os = new PrintStream(clientSocket.getOutputStream());
      is = new DataInputStream(clientSocket.getInputStream());
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host " + host);
    } catch (IOException e) {
      System.err.println("Couldn't get I/O for the connection to the
host "
          + host);
    }

    /*
     * If everything has been initialized then we want to write some
data to the
     * socket we have opened a connection to on the port portNumber.
     */
    if (clientSocket != null && os != null && is != null) {
      try {

        /* Create a thread to read from the server. */
        new Thread(new MultiThreadChatClient()).start();
        while (!closed) {
          os.println(inputLine.readLine().trim());
        }
        /*
         * Close the output stream, close the input stream, close the
socket.
         */
        os.close();
        is.close();
        clientSocket.close();
      } catch (IOException e) {
        System.err.println("IOException:  " + e);
      }
    }
  }

  /*
   * Create a thread to read from the server. (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    /*
     * Keep on reading from the socket till we receive "Bye" from the
     * server. Once we received that then we want to break.
     */
    String responseLine;
    try {
      while ((responseLine = is.readLine()) != null) {
        System.out.println(responseLine);
        if (responseLine.indexOf("*** Bye") != -1)
          break;
      }
      closed = true;
    } catch (IOException e) {
      System.err.println("IOException:  " + e);
    }
  }
}
```

The server

It uses a separate thread for each client. It spawns a new client thread every time a new connection from a client is accepted. This thread opens the input and the output streams for a particular client, it ask the client's name, it informs all clients about the fact that a new client has joined the chat room and, as long as it receive data, echos that data back to all other clients. When the client leaves the chat room, this thread informs also the clients about that and terminates.

```
import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServerSync {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients'
connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new
clientThread[maxClientsCount];

  public static void main(String args[]) {

    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out.println("Usage: java MultiThreadChatServerSync
<portNumber>\n"
          + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }
    /*
     * Open a server socket on the portNumber (default 2222). Note that
we can
     * not choose a port less than 1023 if we are not privileged users
(root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new
client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket,
threads)).start();
            break;
          }
        }
        if (i == maxClientsCount) {
          PrintStream os = new
PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
```

```
 * The chat client thread. This client thread opens the input and the
output
 * streams for a particular client, ask the client's name, informs all
the
 * clients connected to the server about the fact that a new client has
joined
 * the chat room, and as long as it receive data, echos that data back
to all
 * other clients. The thread broadcast the incoming messages to all
clients and
 * routes the private message to the particular client. When a client
leaves the
 * chat room this thread informs also all the clients about that and
terminates.
 */
class clientThread extends Thread {

  private String clientName = null;
  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;

  public clientThread(Socket clientSocket, clientThread[] threads) {
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

  public void run() {
    int maxClientsCount = this.maxClientsCount;
    clientThread[] threads = this.threads;

    try {
      /*
       * Create input and output streams for this client.
       */
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());
      String name;
      while (true) {
        os.println("Enter your name.");
        name = is.readLine().trim();
        if (name.indexOf('@') == -1) {
          break;
        } else {
          os.println("The name should not contain '@' character.");
        }
      }

      /* Welcome the new the client. */
      os.println("Welcome " + name
          + " to our chat room.\nTo leave enter /quit in a new line.");
      synchronized (this) {
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null && threads[i] == this) {
            clientName = "@" + name;
            break;
          }
        }
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null && threads[i] != this) {
            threads[i].os.println("*** A new user " + name
                + " entered the chat room !!! ***");
          }
        }
      }
      /* Start the conversation. */
      while (true) {
        String line = is.readLine();
        if (line.startsWith("/quit")) {
          break;
        }
        /* If the message is private sent it to the given client. */
        if (line.startsWith("@")) {
          String[] words = line.split("\\s", 2);
          if (words.length > 1 && words[1] != null) {
            words[1] = words[1].trim();
            if (!words[1].isEmpty()) {
              synchronized (this) {
                for (int i = 0; i < maxClientsCount; i++) {
                  if (threads[i] != null && threads[i] != this
                      && threads[i].clientName != null
                      && threads[i].clientName.equals(words[0])) {
                    threads[i].os.println("<" + name + "> " +
words[1]);
                    /*
```

```
                     * Echo this message to let the client know the
private
                     * message was sent.
                     */
                    this.os.println(">" + name + "> " + words[1]);
                    break;
                  }
                }
              }
            }
          }
        } else {
          /* The message is public, broadcast it to all other clients.
*/
          synchronized (this) {
            for (int i = 0; i < maxClientsCount; i++) {
              if (threads[i] != null && threads[i].clientName != null)
{
                threads[i].os.println("<" + name + "> " + line);
              }
            }
          }
        }
      }
      synchronized (this) {
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null && threads[i] != this
              && threads[i].clientName != null) {
            threads[i].os.println("*** The user " + name
                + " is leaving the chat room !!! ***");
          }
        }
      }
      os.println("*** Bye " + name + " ***");

      /*
       * Clean up. Set the current thread variable to null so that a
new client
       * could be accepted by the server.
       */
      synchronized (this) {
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] == this) {
            threads[i] = null;
          }
        }
      }
      /*
       * Close the output stream, close the input stream, close the
socket.
       */
      is.close();
      os.close();
      clientSocket.close();
    } catch (IOException e) {
    }
  }
}
```

Running the chat program

Go to the project directory.

Compile both the MultiThreadChatClient.java & MultiThreadChatServerSyn.java by

```
$ javac *.java
```

Run the server.

```
$ java MultiThreadChatServerSyn <port-number>
```

Run (up to 10) clients on same or different machines.

```
$ java MultiThreadChatClient <server-address> <port-number>
```