Fall ’12 CIS 212 Assignment 6 – 110/100 points possible – Due Wednesday, 11-14, 11:59 PM

The goal of this assignment is to build a simple chat client and server which allows communication between multiple clients.


3. [30] Create a new ChatServer project and class in Eclipse. When run, the ChatServer class should create a java.net.ServerSocket which listens for connections on port 7777. Once the server socket is created, print the IP address of the machine running the server to System.out (hint: see the java.net.InetAddress class). The ChatServer project will then need to perform two tasks in parallel:

   1. Repeatedly call the accept() method defined in ServerSocket when run. The accept() method will not return until a client has connected to the socket (i.e., it’s a synchronous call, also known as a blocking call), which is why this loop will need to run in a dedicated thread. When the accept() method returns a connected java.net.Socket instance, add that instance to a Vector of connected sockets. The thread should then sleep for 10 milliseconds to prevent this thread from dominating system resources (hint: see the java.lang.Thread.sleep() method). The thread should keep calling accept() until the server is terminated.

   2. Repeatedly loop through all connected sockets, checking each socket’s input stream for available bytes. Note that Vector is a thread-safe class, so we don’t need to explicitly handle cases in which the Vector is accessed simultaneously from multiple threads. When bytes are available, read those bytes and write them to all sockets (including the socket sending the bytes). If writing to a socket causes an IOException, the socket has been closed remotely so it needs to be closed locally (hint: see the close() method in Socket) and removed from the Vector of connected sockets. After checking all connected sockets for available bytes, the thread should sleep for 10 milliseconds to prevent the thread from dominating system resources. The thread should keep checking sockets for available bytes until the server is terminated.

4. [30] Create a new ChatClient project and class in Eclipse. When run, the ChatClient class should connect to a hardcoded server IP address (port 7777) which can be the local host (hint: again, see the java.net.InetAddress class) using the java.net.Socket class. See the extra credit portion (part 6) for inputting a server IP address. After connecting, your program should write the IP address of the machine running the client plus the text “Joined.” to the socket’s output stream. The ChatClient will then need to loop continually checking for available bytes on the socket’s input stream. When bytes are available, read any available bytes and write them to
System.out. The thread should then sleep for 10 milliseconds to prevent this thread from dominating system resources. The loop should keep checking the socket for available bytes until the client is terminated. You will now be able to test your server and client by running a server and multiple clients in Eclipse, by creating and running executable JAR files, or by running your server/client from the command line.

5. [10] Modify your ChatClient class to extend JFrame. Create a GUI for your ChatClient consisting of a scrollable, uneditable JTextArea (hint: see the JScrollPane class), a JTextField, and a “Send” JButton. Before the socket connects, append the text “Connecting to: ” plus to the IP address of the server to the text area. Once connected, append the text “Connected” to the text area. Add ActionListeners to the text field and button so that pressing enter in the text field or clicking the button causes the text from the text area to be written to the socket’s output stream and the text field cleared. When bytes are available on the socket’s input stream, those bytes should be read and appended to the text area. Add a WindowListener to your frame which responds to the windowClosing(WindowEvent ev) event (hint: see the java.awt.event.WindowAdaptor class) by writing the IP address of the machine running the client plus the text “ Left.” to the socket’s output stream and then closing the socket. Your GUI should look something like:

![Chat Client GUI](image)

Note that you don’t need a dedicated thread to check for input on the client’s socket because Java implicitly creates a GUI thread; the main thread can perform this activity while the GUI thread listens for user events and updates the GUI (we’ll discuss in class).

6. [+10] (Extra credit) Use the javax.swing.JOptionPane class to request a server IP address from the user prior to your ChatClient connecting to your ChatServer.

Zip the ChatServer and ChatClient folders in your Eclipse workspace directory along with your problem-set solution document (PDF or TXT) and upload the .zip file to Blackboard (see Assignment 6 assignment in the Course Documents area).