Crossroads
The ACM Student Magazine

Creating a Simple, Multithreaded Chat System with Java

by ***George Crawford III***

Introduction

In this edition of Objective Viewpoint, you will learn how to develop a simple chat system. The program will demonstrate some key Java programming techniques including I/O, networking, and multithreading.

Chat System Design

The chat system presented here consists of two classes: `ChatServer` and `ChatHandler`. The `ChatServer` class is responsible for accepting connections from clients and providing each client with a dedicated thread for messaging. The `ChatHandler` is an extension of class `Thread`. This class is responsible for receiving client messages and broadcasting those messages to other clients.

ChatServer Code

Listing 1 shows the code for the `ChatServer` class. The class has only one method, `main()`. The first three lines declare three variables: `port`, `serverSocket`, and `socket`. The `port` variable stores the port on which the server will listen for new connections. The default value for the port in this example is `9800`. In the first `try/catch` block, we determine if the user passed any parameters. If so, we attempt to parse the first command line parameter (the only one we care about in this case) using the `Integer.parseInt` method. If the string does not represent an integer, then a

NumberFormatException will be thrown. The catch block simply specifies proper program usage and exits.

In the second try/catch block, the chat server attempts to create a new server socket and begin to accept connections. First, a new ServerSocket object is created with a specific port. The code then enters an endless loop, continuously accepting connections and creating new chat client handlers. The ServerSocket.accept() method waits forever until a new connection is established. When a new connection is detected, a Socket object is created inherently and returned. A new ChatHandler object is then constructed with the newly created socket. Since the ChatHandler is a Thread, we must call the start method to make the chat client code run.

If anything goes awry with either the server socket or client socket, an IOException will be thrown. In this example, we simply print the stack trace. In the finally block, we attempt to close the server socket connection since the loop has been exited.

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

public class ChatServer {
    public static final int DEFAULT_PORT = 9800;
    public static void main(String[] args) {
        int port = DEFAULT_PORT;
        ServerSocket serverSocket = null;
        Socket socket = null;
        try {
            if(args.length > 0)
                port = Integer.parseInt(args[0]);
        } catch(NumberFormatException nfe) {
            System.err.println("Usage: java ChatServer [port]");
            System.err.println("Where options include:");
            System.err.println("\tport the port on which to listen.");
            System.exit(0);
        }
        try {
            serverSocket = new ServerSocket(port);
            while(true) {
```

```
                socket = serverSocket.accept();
                ChatHandler handler = new ChatHandler(socket);
                handler.start();
            }
        } catch(IOException ioe) {
            ioe.printStackTrace();
        } finally {
            try {
                serverSocket.close();
            } catch(IOException ioe) {
                ioe.printStackTrace();
            }
        }
    }
}
```

Listing 1: ChatServer Code

ChatHandler Code

The `ChatHandler` code is shown in Listing 2. In the constructor, we assign the socket the handler we will use and construct the socket input and output streams. The `BufferedReader` and `PrintWriter` classes are used for handling user I/O. The `Reader` classes enable proper handling of bytes and characters. For example, the `BufferedReader` class method `readLine()` will properly convert 8-bit bytes to 16-bit UNICODE characters.

The following line constructs a new `BufferedReader` object:

`in = new BufferedReader(new InputStreamReader(socket.getInputStream()));`

Since the `BufferedReader` constructor requires an object of class `Reader`, we must construct another reader. Since the `Socket.getInputStream()` method returns an object of class `InputStream`, we must create a new `InputStreamReader` to capture the socket input stream. Another advantage of the `BufferedReader` class besides proper character encoding is character buffering. Without buffering, the characters would be read one byte at a time, thus crippling performance.

The construction of the `PrintWriter` is similar in context to the `BufferedReader`.

The `run` method simply captures client input and broadcasts the message to all the other clients. This process continues until the user sends a "quit" message.

The first few lines synchronize on the static `Vector` object `handlers` that contains all the actively connected clients and adds the current object to the list. It is important to synchronize on the list, otherwise other threads that are accessing the list concurrently may miss any newly added clients during a broadcast.

In the `try` block, we enter a continuous loop which is terminated when either the user sends a "/quit" message or an exception is thrown. When a message is received, we propogate the message by iterating through the `ChatHandler` list and sending the message through the appropriate handler socket output stream.

If an exception is thrown or the user sends a "/quit" message, we attempt to close the I/O streams and the socket, and finally remove the current handler from the list.

Listing 2:

```
// ChatHandler.java
//

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Vector;

public class ChatHandler extends Thread {
  static Vector handlers = new Vector( 10 );
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
```

```
    public ChatHandler(Socket socket) throws IOException {
        this.socket = socket;
        in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(
            new OutputStreamWriter(socket.getOutputStream()));
    }
    public void run() {
        String line;
        synchronized(handlers) {
            handlers.addElement(this);
 // add() not found in Vector class
        }
        try {
            while(!(line = in.readLine()).equalsIgnoreCase("/quit")) {
                for(int i = 0; i < handlers.size(); i++) {
                        synchronized(handlers) {
                            ChatHandler handler =
                                (ChatHandler)handlers.elementAt(i);
                        handler.out.println(line + "\r");
                        handler.out.flush();
                        }
                }
            }
        } catch(IOException ioe) {
            ioe.printStackTrace();
        } finally {
            try {
                in.close();
                out.close();
                socket.close();
            } catch(IOException ioe) {
            } finally {
                synchronized(handlers) {
                    handlers.removeElement(this);
                }
            }
        }
    }
}
```

Editor's Note: Many thanks to Dick Seabrook for finding and correcting some errors in the above code listing 2.

Running The Program

To run the server, simply enter the following line at a command prompt:

java ChatServer

The above will open a socket on the default port. Alternatively, you can specify a port like so:

java ChatServer 7900

Also, if you are on a UNIX machine, append an "&" to the above command line so that the server runs in a background process.

Use `telnet` to connect to the server. For example:

telnet machine.somewhere.net 9800

Summary

In this article, you learned how to use the standard Java networking and I/O classes, and also how to handle concurrent I/O safely using threads. As an exercise, try creating a chat GUI, and also expand the current programs to include identifiable users and user groups.

Editor's Addendum
George, there is a curious resemblance between this article and selected publications authored by Merlin Hughes, most notably Java Network Programming (text, 1997) and an online article at http://www.javaworld.com/javaworld/jw-01-1997/jw-01-chat.html. Do you have any comment on this resemblance?

Response from George Crawford:
While I'm surprised at the similarity of our approaches, I'm sure if you study the code closely you will realize that the code for each is unique. The names of the classes are

exactly the same, but appropriate for the type of application being built. It seemed only natural to call a "chat server" ChatServer, and the class that handled the connections for the "chat server" a ChatHandler.

The approach to accepting a socket connection is the same regardless of the type of server app you develop:

1. *create a new server socket*
2. *enter an infinite loop*
3. *accept a connection*
4. *start a new thread to process the connection*
5. *return to 3*

If you look at the code for both server classes, you will find that this is the case. There are very few other ways to do this in Java. This approach is not unique to Java; it's a network programming idiom.

Now, examine the server code again in each case. The author of the other article throws an IOException, whereas I handle the exception and close the server connection. I also attempt to read the first argrument as an int and if that fails, display a message to the user. The other author simply throws an exception with the message in it. Our styles are different here.

If you study the ChatHandler in each case, you'll find that I use different I/O classes than the other author, and also it appears he wrote his class to be extensible, whereas I did not (he uses protected, I use private). The one striking similarity here is that our constructors are structured the same, which is not too odd really if you consider that you must set your socket locally, and the prepare your I/O streams. This too is not uncommon when setting up your client handler.

If you compare the run() and broadcast method for the author to my run() method, you'll see our approach again differs significantly. He uses an Enumeration, I use the Vector. I synchronize on my vector (which in hindsight is unnecessary, since Vector has synchronized methods). I also do a while loop for the client indefinately until the client enters "/quit", while at a glance it looks as though the author's code run's forever (while(true)). The author stops his thread, I do not. I write out a carriage return, he does not. I close my buffers before I close the socket, he does not.

Upon further examination, I can only conclude this resemblance is a fluke. The only glaring similarities I can find are the naming conventions for the classes and the constructor for the client thread.

I should mention that I wrote significant amounts of Java code previous to writing this article. One of those programs was a client/server app much like this for a network programming class in 1996, well before either article.

Resources

1 Sun Microsystems, Inc. The Java 2 Platform API Specification. Sun Microsystems, Inc. January, 1999.

Biography

George Crawford III is a software engineer at MPI Software Technology, Inc. and completing his M.S. degree in Computer Science at Mississippi State University. His technical areas of interest include software design, distributed object technology, Java programming, games programming, and DirectX/Direct3D programming.