Building an Internet chat system

Multithreaded client/server chat -- the Java way

By Merlin Hughes

JavaWorld | Jan 1, 1997 12:00 AM PT

You may have seen one of the many Java-based chat systems that have popped up on the Web. After reading this article, you'll understand how they work -- and know how to build a simple chat system of your own.

This simple example of a client/server system is intended to demonstrate how to build applications using just the streams available in the standard API. The chat uses TCP/IP sockets to communicate, and can be embedded easily in a Web page. For reference, we provide a sidebar explaining Java network programming components that are relevant to this application. If you're still getting up to speed, take a look at the sidebar first. If you're already well-versed in Java, though, you can jump right in and simply refer to the sidebar for reference.

Building a chat client

We start with a simple graphical chat client. It takes two command-line parameters -- the server name and the port number to connect to. It makes a socket connection and then opens a window with a large output region and a small input region.

The ChatClient interface

After the user types text into the input region and hits Return, the text is transmitted to the server. The server echoes back everything that is sent by the client. The client displays everything received from the server in the output region. When multiple clients connect to one server, we have a simple chat system.

Class ChatClient

This class implements the chat client, as described. This involves setting up a basic user interface, handling user interaction, and receiving messages from the server.

```
import java.net.*;
import java.io.*;
import java.awt.*;
public class ChatClient extends Frame implements Runnable {
  // public ChatClient (String title, InputStream i, OutputStream o) ...
  // public void run () ...
  // public boolean handleEvent (Event e) ...
  // public static void main (String args[]) throws IOException ...
}
```

The `ChatClient` class extends `Frame`; this is typical for a graphical application. We implement the `Runnable` interface so that we can start a `Thread` that receives messages from the server. The constructor performs the basic setup of the GUI, the `run()` method receives messages from the server, the `handleEvent()` method handles user interaction, and the `main()` method performs the initial network connection.

```
  protected DataInputStream i;
  protected DataOutputStream o;
  protected TextArea output;
  protected TextField input;
  protected Thread listener;
  public ChatClient (String title, InputStream i, OutputStream o) {
    super (title);
    this.i = new DataInputStream (new BufferedInputStream (i));
    this.o = new DataOutputStream (new BufferedOutputStream (o));
    setLayout (new BorderLayout ());
    add ("Center", output = new TextArea ());
    output.setEditable (false);
    add ("South", input = new TextField ());
    pack ();
    show ();
    input.requestFocus ();
    listener = new Thread (this);
    listener.start ();
  }
```

The constructor takes three parameters: a title for the window, an input stream, and an output stream. The `ChatClient` communicates over the specified streams; we create buffered data streams *i* and *o* to provide efficient higher-level communication facilities over these streams. We then set up our simple user interface, consisting of the `TextArea` *output* and the `TextField` *input*. We layout and show the window, and start a `Thread` *listener* that accepts messages from the server.

```
public void run () {
    try {
      while (true) {
        String line = i.readUTF ();
        output.appendText (line + "\n");
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    } finally {
      listener = null;
      input.hide ();
      validate ();
      try {
        o.close ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }
```

When the listener thread enters the run method, we sit in an infinite loop reading `String`s from the input stream. When a `String` arrives, we append it to the output region and repeat the loop. An `IOException` could occur if the connection to the server has been lost. In that event, we print out the exception and perform cleanup. Note that this will be signalled by an `EOFException` from the `readUTF()` method.

To clean up, we first assign our *listener* reference to this `Thread` to `null`; this indicates to the rest of the code that the thread has terminated. We then hide the input field and call `validate()` so that the interface is laid out again, and close the `OutputStream` *o* to ensure that the connection is closed.

Note that we perform all of the cleanup in a `finally` clause, so this will occur whether an `IOException` occurs here or the thread is forcibly stopped. We don't close the window immediately; the assumption is that the user may want to read the session even after the connection has been lost.

```
public boolean handleEvent (Event e) {
    if ((e.target == input) && (e.id == Event.ACTION_EVENT)) {
      try {
        o.writeUTF ((String) e.arg);
        o.flush ();
      } catch (IOException ex) {
        ex.printStackTrace();
        listener.stop ();
      }
      input.setText ("");
      return true;
    } else if ((e.target == this) && (e.id == Event.WINDOW_DESTROY)) {
      if (listener != null)
        listener.stop ();
      hide ();
      return true;
    }
    return super.handleEvent (e);
  }
```

In the `handleEvent()` method, we need to check for two significant UI events:

The first is an action event in the `TextField`, which means that the user has hit the Return key. When we catch this event, we write the message to the output stream, then call `flush()` to ensure that it is sent immediately. The output stream is a `DataOutputStream`, so we can use `writeUTF()` to send a `String`. If an `IOException` occurs the connection must have failed, so we stop the *listener* thread; this will automatically perform all necessary cleanup.

The second event is the user attempting to close the window. It is up to the programmer to take care of this task; we stop the *listener* thread and hide the `Frame`.

```
public static void main (String args[]) throws IOException {
    if (args.length != 2)
      throw new RuntimeException ("Syntax: ChatClient <host> <port>");
    Socket s = new Socket (args[0], Integer.parseInt (args[1]));
    new ChatClient ("Chat " + args[0] + ":" + args[1],
                    s.getInputStream (), s.getOutputStream ());
  }
```

The `main()` method starts the client; we ensure that the correct number of arguments have been supplied, we open a `Socket` to the specified host and port, and we create a `ChatClient` connected to the socket's streams. Creating the socket may throw an exception that will exit this method and be displayed.

Building a multithreaded server

We now develop a chat server that can accept multiple connections and that will broadcast everything it reads from any client. It is hardwired to read and write `String`s in UTF format.

There are two classes in this program: the main class, `ChatServer`, is a server that accepts connections from clients and assigns them to new connection handler objects. The `ChatHandler` class actually does the work of listening for messages and broadcasting them to all connected clients. One thread (the main thread) handles new connections, and there is a thread (the `ChatHandler` class) for each client.

Every new `ChatClient` will connect to the `ChatServer`; this `ChatServer` will hand the connection to a new instance of the `ChatHandler` class that will receive messages from the new client. Within the `ChatHandler` class, a list of the current handlers is maintained; the `broadcast()` method uses this list to transmit a message to all connected `ChatClient`s.

Class ChatServer

This class is concerned with accepting connections from clients and launching handler threads to process them.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class ChatServer {
  // public ChatServer (int port) throws IOException ...
  // public static void main (String args[]) throws IOException ...
}
```

This class is a simple standalone application. We supply a constructor that performs all of the actual work for the class, and a `main()` method that actually starts it.

```
  public ChatServer (int port) throws IOException {
    ServerSocket server = new ServerSocket (port);
    while (true) {
      Socket client = server.accept ();
      System.out.println ("Accepted from " + client.getInetAddress ());
      ChatHandler c = new ChatHandler (client);
      c.start ();
    }
  }
```

This constructor, which performs all of the work of the server, is fairly simple. We create a `ServerSocket` and then sit in a loop accepting clients with the `accept()` method of `ServerSocket`. For each connection, we create a new instance of the

`ChatHandler` class, passing the new `Socket` as a parameter. After we have created this handler, we start it with its `start()` method. This starts a new thread to handle the connection so that our main server loop can continue to wait on new connections.

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

The `main()` method creates an instance of the `ChatServer`, passing the command-line port as a parameter. This is the port to which clients will connect.

Class ChatHandler

This class is concerned with handling individual connections. We must receive messages from the client and re-send these to all other connections. We maintain a list of the connections in a

`static`

`Vector`.

```
import java.net.*;
import java.io.*;
import java.util.*;
public class ChatHandler extends Thread {
  // public ChatHandler (Socket s) throws IOException ...
  // public void run () ...
}
```

We extend the `Thread` class to allow a separate thread to process the associated client. The constructor accepts a `Socket` to which we attach; the `run()` method, called by the new thread, performs the actual client processing.

```
  protected Socket s;
  protected DataInputStream i;
  protected DataOutputStream o;
  public ChatHandler (Socket s) throws IOException {
    this.s = s;
    i = new DataInputStream (new BufferedInputStream (s.getInputStream ()));
    o = new DataOutputStream (new BufferedOutputStream (s.getOutputStream ()));
  }
```

The constructor keeps a reference to the client's socket and opens an input and an output stream. Again, we use buffered data streams; these provide us with efficient I/O and methods to communicate high-level data types -- in this case, `String`s.

```
protected static Vector handlers = new Vector ();
  public void run () {
    try {
      handlers.addElement (this);
      while (true) {
        String msg = i.readUTF ();
        broadcast (msg);
      }
    } catch (IOException ex) {
      ex.printStackTrace ();
    } finally {
      handlers.removeElement (this);
      try {
        s.close ();
      } catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
  // protected static void broadcast (String message) ...
```

The `run()` method is where our thread enters. First we add our thread to the `Vector` of `ChatHandler`s *handlers*. The *handlers* `Vector` keeps a list of all of the current handlers. It is a `static` variable and so there is one instance of the `Vector` for the whole `ChatHandler` class and all of its instances. Thus, all `ChatHandler`s can access the list of current connections.

Note that it is very important for us to remove ourselves from this list afterward if our connection fails; otherwise, all other handlers will try to write to us when they broadcast information. This type of situation, where it is imperative that an action take place upon completion of a section of code, is a prime use of the `try ... finally` construct; we therefore perform all of our work within a `try ... catch ... finally` construct.

The body of this method receives messages from a client and rebroadcasts them to all other clients using the `broadcast()` method. When the loop exits, whether because of an exception reading from the client or because this thread is stopped, the `finally` clause is guaranteed to be executed. In this clause, we remove our thread from the list of handlers and close the socket.

```
protected static void broadcast (String message) {
    synchronized (handlers) {
      Enumeration e = handlers.elements ();
      while (e.hasMoreElements ()) {
        ChatHandler c = (ChatHandler) e.nextElement ();
```

```
          try {
            synchronized (c.o) {
              c.o.writeUTF (message);
            }
            c.o.flush ();
          } catch (IOException ex) {
            c.stop ();
          }
        }
      }
    }
```

This method broadcasts a message to all clients. We first synchronize on the list of handlers. We don't want people joining or leaving while we are looping, in case we try to broadcast to someone who no longer exists; this forces the clients to wait until we are done synchronizing. If the server must handle particularly heavy loads, then we might provide more fine-grained synchronization.

Within this synchronized block we get an `Enumeration` of the current handlers. The `Enumeration` class provides a convenient way to iterate through all of the elements of a `Vector`. Our loop simply writes the message to every element of the `Enumeration`. Note that if an exception occurs while writing to a `ChatClient`, then we call the client's `stop()` method; this stops the client's thread and therefore performs the appropriate cleanup, including removing the client from *handlers*.

Page 2 of 2

Note that the `writeUTF()` method is not synchronized, so we must explicitly perform synchronization to prevent other threads from writing to the stream at the same time.

Wrapping up

Multithreading is essential for servers with any sophistication. In this article, we have developed a multithreaded broadcast chat server and a simple graphical client. We used data streams to read and write messages in UTF format. Although this certainly makes more sense than breaking everything down into bytes, there is one drawback to this implementation: We can communicate only single `String` messages between clients. If we wish to communicate different information, then we must change both the client *and* the server. This can be a serious limitation when we want the server to perform a more generalized purpose, or when we want to extend the system to handle more than just text.

A better solution

The solution to this problem is to develop an application-layer protocol that defines a more powerful level of dialog between a client and the server. Typically, such a protocol would encapsulate messages between the client and server by transmitting a header and a body with each message; the header identifies the type of the message and the length of the message body, and the body contains the actual message information. This concept is similar to the headers that can be attached to HTTP requests and responses.

By providing this meta-information, the client and server system can be extended easily. We can add new message types to the protocol without affecting existing code; if a client does not understand a message of a new type, it can simply discard the message. Furthermore, we can introduce much more powerful messages; a message of type *MSG_TEXT* may consist of simply a `String`, but a message of type *MSG_SCRIBBLE* may consist of a sequence of `Point`s. If the application protocol is appropriately defined, then the server does not need to understand these messages; the header identifies the length of each message, and so the server can relay messages between clients without understanding the format of the message bodies.

Message headers also can be used for other purposes. For example, they can be used to identify a particular client to whom a message should be relayed; instead of a broadcast-only server, we can extend the server to provide unicast and multicast services. Headers also can identify control information coming back from the server, so the server can provide notification of events such as clients joining and leaving.

Implementation

When it comes to implementing such an application-layer protocol, there are two main options. The protocol can be supported explicitly by all applications, so that when a client is sending a message, it manually inserts the appropriate header information. An alternative, however, and much more elegant solution, is to abstract the protocol behind a set of stream classes. The specifics of header construction and insertion can be handled automatically by the stream classes, and the client is then left with much the same interface as before: Clients write messages to a stream, but instead of flushing the stream, they call a method that attaches appropriate headers and sends the encapsulated message.

The details of such a set of stream classes obviously will vary between different applications; in the book *Java Network Programming*, we develop, among other things, a generic set of stream classes that provide just this type of function.

The chat system in actionMerlin Hughes has extensive experience in cryptography, networking, and computer graphics and has published and lectured on these topics. He is the VP of Technology at Prominence Dot Com, a North Carolina company formed to create interactive products for the Web. Prominence is currently in development of a series of Java-based customer service and marketing applications to complement corporate Web strategies.