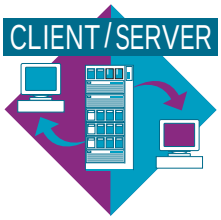
Applets and Servlets: A Smooth Blend



By Greg Flurry and Jeff Jilg

What problems are encountered in Java client/server programming? How can you use servlets to reduce communications overhead? Can applets and servlets be used to complement each other in a robust design? This article addresses these issues and more.

Java™ servlets require some careful planning and consideration during the design and development stage. Some issues such as initialization, multithreading, and synchronization using a Java applet/servlet-based chat application are described in this article. The article also includes a detailed breakdown of the chat application programming logic, coupled with a discussion of design choices.

We used Windows 95 and VisualAge™ for Java for developing this application, so we included a discussion of the development and debugging environment. After we developed the applications, the servlet portion was executed on AIX® to take advantage of the robust Java engine provided there. Future development could take place on AIX since the VisualAge for Java environment was demonstrated at the March 1998 JavaOne™ Conference, although product availability was not announced.

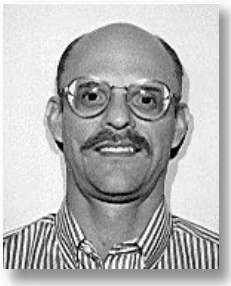
Java servlets appear destined to fill the role of CGI bin in many future Web server environments. Although many articles describing the importance of servlets and basic servlet development are available, most introductory articles ignore many of the interesting problems in designing and developing servlets. This article skips servlet basics and proceeds directly to some "intermediate" issues related to servlet design, and offers some tips on developing and debugging servlets.[1]

Design Considerations

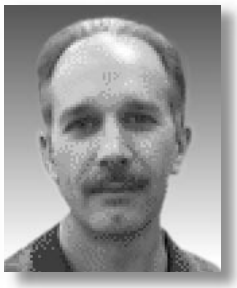
The goal of our application called Chat is to allow several users of a Web browser to enter information into a "chat room" and enable all members of that chat room to see everything that any member enters. This article examines the servlet portion of a Web-based chat application that was implemented using a Java applet and a Java servlet.

Why use a servlet in this application? Figure 1 shows the Chat design in which each individual user fires up an applet running in a browser and wants to chat. But where does the applet get the list of Chat participants? Most likely that list comes from a server.

Since each user's applet must connect to all other applets, the number of required connections $((n^2-n)/2)$ increases exponentially. For example, this design for four users requires six connections. The applet for Chat design must implement the complexity of discovering new connections and maintaining them, which makes it potentially unsuitable for low-end clients. This design may also create network bandwidth problems.



Greg Flurry



Jeff Jilg

[1] For basic information about servlets, see "Using Java Servlets on AIX" in the March 1998 issue of AIXpert.

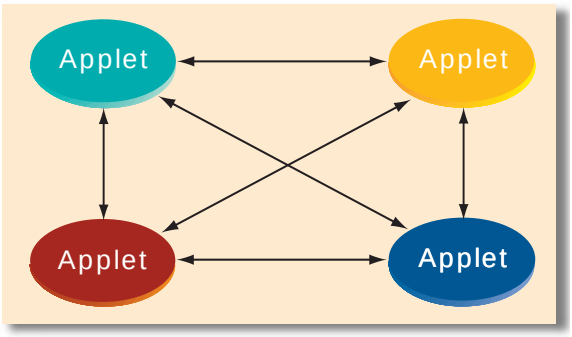


Figure 1. Applet-only Chat application

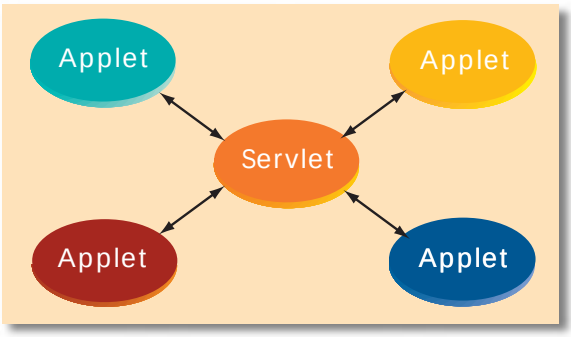


Figure 2. Applet-Servlet Chat application

Now consider the Chat design shown in Figure 2. In this example, each applet connects only to a central server; there is no need for the applet to even know about other participants. The applet is now light-weight and well suited for low-end clients. And there is only a single connection for each Chat participant—the number of connections *(n)* increases only linearly. Four clients require only four connections in this design. This decreases network bandwidth consumption, especially for large groups. For 10 users, the design uses just 10 connections compared to 45 connections required for the applet-only Chat.

Other advantages are also apparent. For example, the Chat server (in this case, the servlet running in a Web server) could support multicast datagram sockets. These sockets allow the sender to send a datagram to multiple receivers in a single transmission for potential further bandwidth reduction. This design uses unicast connection-based sockets. The Chat server could also support multiple chat rooms, maintain a log of all the chat room activity, or even perform load balancing transparently to the clients. However, our example does not implement these functions.

The Chat Application

The Chat application uses applets and a servlet as clients and server, respectively, to enable messages to be passed back and forth by users. A similar design could be incorporated into such network computing applications as remote debugging or system management consoles, or even Internet telephony (for example, for multiparty calls).

The following sections provide an overview of the Chat architecture, the states of the architecture, and termination of the application. The Java code in this article can be found in separate files on the CD-ROM. You can also copy and paste from Adobe Acrobat® by choosing the "ABC" icon and highlighting the code in Acrobat with your mouse.

Architecture Overview

At a high level, the Chat application runs an applet in a Web browser that presents a graphical user interface (GUI), shown in Figure 3. The GUI allows the user to enter text to send to the Chat server and to

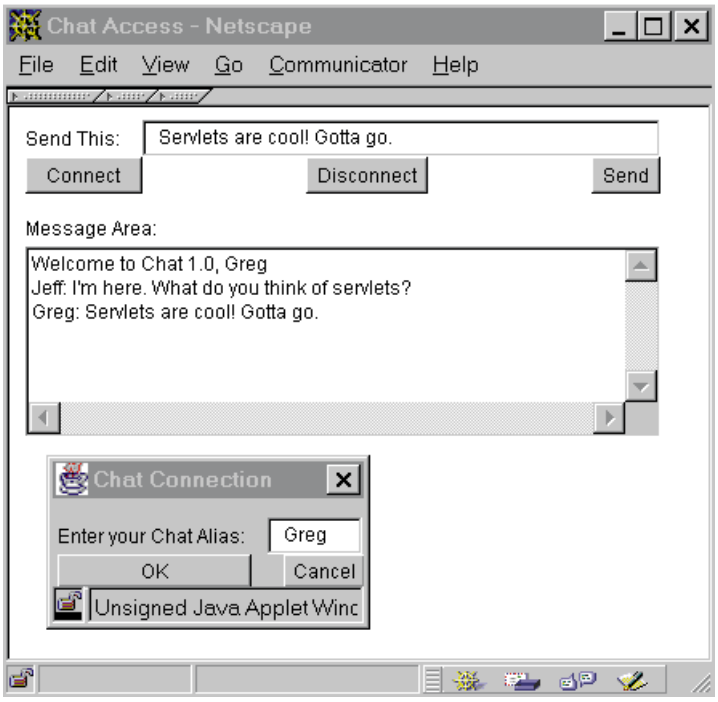


Figure 3. An Applet and its connect dialog

display text received from the Chat server. A connection-based socket handles communication between the applet and the servlet. The primary applet thread sends input from the user to the Chat server. For sending and receiving messages simultaneously, the applet spawns a second thread to listen for output from the servlet.

The servlet also involves multiple threads. The primary thread runs the standard `HttpServlet.doGet()` method, which is invoked for each instance of `ChatApplet` that connects to the Chat server. The `doGet()` method spawns a new thread that interacts with the corresponding applet on the client side; the `doGet()` thread then returns (ends). Any time a server-side listener receives input from a client, it sends that input, prefaced by an alias unique to that client, to all the existing client-side listeners. Details of this operation are described below.

Initialization

Figure 4 shows the HTML file that a user references to begin the Chat application. This file typically would be stored on the same server as the Chat servlet, although that is not required.

The parameters used to indicate the server hostname and port number allow the Chat administrator to easily modify this information as required. This also proved to be a useful debugging aid because it was easy to switch between testing the applet against the JavaSoft™ *srun* servlet environment and the Lotus Go™ 4.6.1 servlet environment.

Figure 5 shows part of the code for the `ChatApplet` class that runs in the primary applet thread. The GUI is constructed using VisualAge for Java with standard Abstract Windowing Toolkit (AWT) classes and the Java 1.1 event model (this code is not shown).

```
<html>
<title>Chat</title>
<body>
<h1>Chat 1.0</h1>
<applet code="test.ChatApplet.class" width=400 height=200>
<param name="hostname" value="flurry">
<param name="hostport" value="80">
</applet>
</body>
</html>
```

Figure 4. Chat application HTML file

```
public class ChatApplet extends Applet implements
java.awt.event.ActionListener {
  ChatListener chatListen = null;
  Socket client = null;
  BufferedReader in = null;
  InputStreamReader inISR = null;
  private Dialog ivjDialog1 = null;
  private TextArea ivjChatRecord = null;
  private TextField ivjAlias = null;
  private TextField ivjToServer = null;
  PrintWriter out = null;
  String host = "flurry";
  int port = 80;

  public void init() {
    super.init();
    try {
```

Figure 5. ChatApplet class code segments (continued on following page)

```
      System.out.println("ChatApplet version "+version+" started");

      /* lots of GUI initialization code not shown */

      // process the applet parameters
      String hostname = getParameter("hostname");
      String hostport = getParameter("hostport");
      if (hostname != null) { host = hostname; }
      if (hostport != null) { port = new Integer(hostport).intValue(); }
    } catch (java.lang.Throwable ivjExc) {
      handleException(ivjExc);
    }
  }

  public void connect( ) {

    String newPort = null;
    URL chatURL = null;

    // get alias
    String alias = ivjAlias.getText();
    ivjDialog1.dispose();

    System.out.println("Entering connect()");
    try {
      // point to server
      chatURL = new URL("http", host, port, "/servlet/ChatServlet");
    } catch (MalformedURLException e) {
      System.out.println("URL problem: " + e);
    }

    System.out.println("Formed URL");
    try {
      // contact server
      InputStream serverIn = (InputStream) chatURL.getContent();
      BufferedReader bSI = new BufferedReader(new
InputStreamReader(serverIn));
      System.out.println("Contacted server");
      newPort = bSI.readLine();
      int thePort = new Integer(newPort).intValue();
      System.out.println("Contacted servlet and got a response of " +
newPort);

      if (thePort == 0) {
        // can't connect to server
        ivjChatRecord.append("Sorry, cannot connect to server; try later.\n");
        return;
      }

      // attach to new server port
      client = new Socket(host, thePort);

      // get i/o streams
      inISR = new InputStreamReader(client.getInputStream());
      in = new BufferedReader(inISR);
      out = new PrintWriter(client.getOutputStream(),true);

    } catch (IOException e) {
```

Figure 5. ChatApplet class code segments (continued on following page)

```
        System.out.println("IOException:  " + e);
      }
      // send the alias string
      out.println(alias);

      // establish a listener for the server
      chatListen = new ChatListener(in, ivjChatRecord);
      chatListen.start();

      return;
    }

    public void sendText( ) {
      String request = ivjToServer.getText();
      out.println(request);
      return;
    }

    public void disconnect( ) {

      // stop the server
      out.println("QUIT");

      // now wait for the listener to quit
      chatListen.stopListening();

      // close all the things
      try {
        in.close();
        out.close();
        client.close();
      } catch (IOException e) {
        System.out.println("IOException:  " + e);
      }

      return;
    }

  }
```

Figure 5. ChatApplet class code segments

The GUI (Figure 3) presents Connect, Disconnect, and Send buttons, a `TextField` for user input to be sent to the Chat server, and a `TextArea` that displays all output sent from the Chat server.

By pressing Connect, a Chat connection `Dialog` (Figure 3) appears containing a `TextField` where the user enters an alias to identify that user in the chat room; the `Dialog` also contains an OK button. Pressing OK invokes `ChatApplet.connect()`, shown in Figure 5. First, `connect()` retrieves the alias for the user, then creates a URL using the applet parameters. These parameters point to the Chat servlet; `connect()` gets the content of the URL if the servlet has been initialized. If the Chat servlet has not been initialized, the servlet environment invokes the `init()` method for the `ChatServlet` class (Figure 6) that runs in the primary servlet thread.

The `init()` method in this example sets the length of a `PrintWriter` array and an `int` array. The array of `PrintWriters` communicates with the clients that attach to the server. The array of `ints` holds the numbers of the ports used in the socket connections. Since `ChatServlet` also uses this array as an indication of whether clients are attached, it is initialized to all

zeros. In addition, `init()` sets up the base number of the ports used for communications between the clients and the server.

If `ChatServlet` has been initialized, the servlet environment invokes the `doGet()` method. `ChatServlet.doGet()` first sets up for HTTP communication with the applet, then searches the port array for an open slot. Note that searching through the array and reserving a slot in the array are "synchronized."

Synchronization is critical to ensure the integrity of the port and `PrintWriter` arrays. Given the nature of the servlet environment, it is possible for multiple instances of `ChatServlet.doGet()` to run simultaneously, each trying to access the array of ports to find an open slot and reserve a slot. Without synchronization, two instances could end up finding the same empty slot, or possibly not finding an open slot—even though one really exists.

It is even more likely that one or more server-side listeners could be accessing the `PrintWriter` array and then try to use a `PrintWriter` simultaneously—with unpredictable (but most likely unpleasant) results. Although it is possible to synchronize the entire `doGet()` method, synchronizing only the sections dealing with the arrays provides better responsiveness. Synchronization is critical to maintain integrity of the `ChatServlet` application.

Following the search, `ChatServlet.doGet()` returns a string that contains either the port number to use for communications with the server, or zero, indicating that no more open slots are available for communications with the server. If `ChatServlet` has an open slot, `doGet()` starts a new thread running the `ServerListener` class to interact with the Chat applet.

It is necessary to spawn a separate thread to listen to the applet. Whenever a servlet `service()` method is invoked, the servlet environment spawns a new thread in which to run the method. During testing, however, we found that a Web server does not complete the HTTP transaction with the Web browser until the service method returns. The Web browser expects more input from

```
public
class ChatServlet extends HttpServlet {

  int basePort = 0;
  int chatClient = 0;
  PrintWriter clientOut[] = null;
  int clientPort[] = null;

  String version = "1.03";

  public void init(ServletConfig config)
       throws ServletException
  {

    super.init(config);

    // here perhaps would set up the maximum number of clients
    int maxLength = 4;
    // initialize the servlet state
    clientOut = new PrintWriter[maxLength];
    clientPort = new int[maxLength];
    for (int i=0; i<clientOut.length; i++) {
      clientOut[i] = null;
      clientPort[i] = 0;
    }
```

Figure 6. ChatServlet class code (continued on following page)

```
    // set up the base port number for communications
    basePort = 9000;
  }

  public void destroy() {
    log("about to destroy");
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
       throws ServletException, IOException
  {

    int me;

    // set up for returning information
    log("entering doGet");
    ServletOutputStream bout = res.getOutputStream();
    res.setContentType("text/plain");
    // get client number and port (find unused output slot)
    synchronized (clientPort) {
      for (me=0; me<clientPort.length; me++) {
        if (clientPort[me] == 0) {break;}
      }
      // if found an open slot
      if (me < clientOut.length) {
        // reserve the slot
        clientPort[me] = basePort + me;
      }
    }

      // if did not find an open slot
    if (me == clientOut.length) {
      // indicate can't support another client
      log("doGet, overbooked");
      bout.println(""+0);
    } else {
      // send port back to applet
     log("doGet, accepting another client");
     bout.println(""+clientPort[me]);
      // start a new server thread to listen to client
      ServerListener chatter = new ServerListener(me, clientPort, clientOut,
this);
      log("doGet, about to start chatter");
      chatter.start();
    }
    // finish off
    bout.flush();
    return;
  }
}
```

Figure 6. ChatServlet class code

the Web server and thus "spins." We chose to start another thread that interacts with the client so that the servlet's service method can return, thus completing the HTTP transaction as far as the Web server and Web browser are concerned.

Notice the `ChatApplet.connect()` call to `URL.getContent()` in which `URL.getContent()` returns an `Object`. The actual class is really a subclass of `Object`, specific to the content type. Testing showed that the actual class returned depends on the environment in which `ChatApplet` runs. IBM's VisualAge for Java and Sun's Java Development Kit (JDK) for Windows™ returned the same class, but Netscape™ Communicator returned a different class. Fortunately, both were subclasses of `InputStream`. We have not tested this in other browsers, and not all will necessarily return subclasses of `InputStream`, although it seems likely they would do so. Based on the observation, the `Object` is cast as an `InputStream`.

The `InputStream` is used to create a `BufferedReader` so that `connect()` can easily read a string returned from the Chat servlet. As mentioned above, the string is either "0", which indicates a failure, or a port number for contacting the server. If it receives a port number, `connect()` attaches to the server socket and retrieves an `InputStream` and `OutputStream` for receiving strings from and sending strings to the server. It then sends to the server the alias retrieved from the `Dialog`. Finally, `connect()` creates a new thread running a `ChatListener` class (shown in Figure 7) which receives input from the server using the `BufferedReader` and then echos it to the `TextArea`. When the `ChatServlet` thread instantiates the `ServerListener` thread (see Figure 8), it passes the port array and `PrintWriter` array to the constructor and an index into the array for this particular instance. It also passes an object reference to the servlet, which allows events to be logged in the `ServerListener` as needed (see more on this below). Once it has begun running, `ServerListener.run()` creates a `ServerSocket` listening on the port indicated by the index, and waits for `ChatApplet` to attach to the socket. After accepting the attachment, `ServerListener` gets the input and output streams necessary for communication. It places the `PrintWriter` created using the output stream into the `PrintWriter` array so that it can be accessed by all instances of `ServerListener`; this action must be synchronized for the reasons described above. `ServerListener` then reads the alias sent by `ChatApplet` and sends a "welcome" message to the `ChatListener` instantiated by the `ChatApplet`. `ServerListener` then begins its "steady state" operation.

Steady State

After the initialization activities, the `ChatApplet` thread waits for input from the user so it can send text to the `ServerListener` thread. The `ServerListener` thread waits for the text from the `ChatApplet` thread and echos input to all the `ChatListener` threads in the session. A `ChatListener` thread receives text from multiple `ServerListeners` and echos that text to the `ChatApplet TextArea`.

> ***VAJava is an excellent integrated development environment for developing Java applications, and especially good for developing user interfaces.***

To send text, the user must type into the `TextField` (Figure 3) presented by the `ChatApplet` and press the "Send" button, which invokes `ChatApplet.sendText()`, shown in Figure 5. Then, `sendText()`retrieves the text from the `TextField` and writes it to the socket's output stream, sending it to the corresponding `ServerListener` instance.

The `ServerListener.run()` method (Figure 7) waits for input from the `ChatApplet`. It receives the string and checks to see if the user wants to `QUIT`. If not, `run()` sends the received string to all the `ChatListeners` connected to the server by entering a loop and checking for valid `PrintWriters`. This action must be synchronized for the reasons described above. Then, `run()` writes the string to

```
public class ChatListener extends Thread {
  BufferedReader inStream = null;
  TextArea outTarget = null;

  boolean done = false;

  public ChatListener (BufferedReader in, TextArea ta) {
    inStream = in;
    outTarget = ta;
  }

  /* NOTE: other constructor forms not shown, i.e., the form
     with no parameters required for compatibility with
     JavaBeans
     */

  public void run( ) {

    // set up to run as long as user wants
    done = false;

    // if sufficient information to proceed
    if ((inStream != null) && (outTarget != null)) {
      // run forever
      while (go) {
        // wait for input
        try {
          String message = inStream.readLine();
          // write it out
          outTarget.append(message + "\n");
          // check for user finished
          if (message.startsWith("Goodbye")) {
            // indicate we are finished
            done = true;
          }
        } catch (IOException e) {
          System.out.println("Server socket problem in ChatListener: " +  e);
        }
      }
    }


// indicate finished
        done = true;

    return;
  }

  public void stopListening() {

    // wait for the run method to finish
    while(!done) {}

    return;
  }
}
```

Figure 7. ChatListener class code

each valid PrintWriter. Notice that the synchronization object is the port array rather than the PrintWriter array, yet we check for PrintWriters before writing to a client. The servlet must synchronize on the port array because that is the mechanism used to accept or reject clients. Due to nondeterministic scheduling, the port could be allocated, yet the socket connection is not yet made; thus, the PrintWriter is not yet instantiated.

The ChatListener.run() method (Figure 8) waits for input from ServerListeners. It simply gets a string from the input stream and appends it to the TextArea. The run() method also checks each received string for the indication to terminate. Since the ChatListener is the only entity writing to the ChatApplet TextArea, there are no synchronization issues.

Termination

The steady state operation described earlier continues until the user presses the Disconnect button, shown in Figure 3. This invokes ChatApplet.disconnect() (Figure 5), which sends the QUIT string to the server indicating that it wants to terminate the communication. disconnect() then waits for the ChatListener to stop listening and terminate by calling ChatListener.stopListening(), which

```
class ServerListener extends Thread {
  int me;
  int myPort;
  PrintWriter clientOut[] = null;
  int clientPort[] = null;
  HttpServlet servlet = null;

  ServerSocket ssClient = null;
  Socket socket = null;
  InputStreamReader inISR = null;
  BufferedReader in = null;
  String version = "1.04";

  public ServerListener (int index, int[] port, PrintWriter[] out, HttpServlet
it) {
    super();

    // set up information
    it.log("ServerListener version "+version+": port="+port);
    me = index;
    clientPort = port;
    myPort = clientPort[me];
    clientOut = out;
    servlet = it;
  }

 public void run()
  {

    boolean keepon = true;

    ServerSocket ssClient = null;
    Socket socket = null;
    InputStreamReader inISR = null;
    BufferedReader in = null;

    try {
      // wait for a client to connect
```

Figure 8. ServerListener class code (continued on following page)

```
servlet.log("ServerListener "+version+ " waiting for client connection to:
"+myPort);
      // get server socket
      ssClient = new ServerSocket(myPort);
      servlet.log("ServerListener: got ServerSocket("+myPort+")");

      // wait for a client to connect
      servlet.log("ServerListener: waiting for client connection client "+me);
      socket = ssClient.accept();
      servlet.log("ServerListener: got client connection client");

      // get input and output streams (output with autoflush)
      inISR = new InputStreamReader(socket.getInputStream());
      in = new BufferedReader(inISR);
      synchronized (clientPort) {
        clientOut[me] = new PrintWriter(socket.getOutputStream(),true);
      }

      // get alias
      String alias = in.readLine();

      // send welcome message
      clientOut[me].println("Welcome to Chat 1.0, " + alias);

      // service messages
      while (keepon) {

        // get input from client
        servlet.log("ServerListener: waiting on client "+me);
        String request = in.readLine();
       // if quitting
        if (request.startsWith("QUIT")) {
          // finish up
          servlet.log("ServerListener: about to send to client");
          clientOut[me].println("Goodbye, " + alias);
          servlet.log("ServerListener: sent QUIT to "+me);
          keepon = false;
        } else {
          // send input to all clients
          synchronized (clientPort) {
            for (int i=0; i<clientOut.length; i++) {
              // if there is a client in this slot
              if (clientOut[i] != null) {
                // send to client
                servlet.log("ServerListener: about to send to client "+i);
                clientOut[i].println(alias + ": " + request);
                servlet.log("ServerListener: sent to client");
              }
            }
          }
        }
      }

      // close everything
      servlet.log("ServerListener: about to quit "+me);
      synchronized (clientOut) {
        clientOut[me].close();
        clientOut[me] = null;
```

Figure 8. ServerListener class code (continued on following page)

```
        clientPort[me] = 0;
        }
        in.close();
        inISR.close();
        socket.close();
        ssClient.close();

      } catch (IOException e) {
        servlet.log("ServerListener IOException: " +  e);
      }
    }

}
```

Figure 8. ServerListener class code

does not return until the `ChatListener` is finished.

The `ServerListener.run()` method (Figure 7) receives the `QUIT` string to indicate that the user wants to disconnect. The `ServerListener` sends a "goodbye" message to only the terminating `ChatListener`. This string ensures that the `ChatListener's` socket read is satisfied, allowing it to terminate. The `ServerListener` closes the output stream, then frees its slot in both the port and `PrintWriter` arrays. These actions must be synchronized. The `ServerListener` then closes the input stream and the sockets, releasing them for the next user. Finally, it returns, thereby terminating the thread.

Once the `ChatListener.run()` method (Figure 8) receives the "goodbye" message from its corresponding `ServerListener`, it then indicates it is done. `ChatListener` returns, terminating the thread. Now `ChatApplet.disconnect()` (Figure 5) closes the socket input and output streams and the socket itself. From the user standpoint, the Chat application is now finished.

Developing the Chat Application

We coded and tested both the applet and the servlet in Windows 95. We used IBM's VisualAge for Java (VAJava) and Sun's JDK tools for developing and testing the applet. VAJava is an excellent integrated development environment for developing Java applications, and especially good for developing user interfaces. VAJava offers excellent debugging facilities that allow breakpoints, real-time examination of values, and so on. VAJava was demonstrated on AIX at the JavaOne Conference in March 1998, although no general availability was announced. You will probably want to execute the servlet on AIX to take advantage of the performance on a server. We think development on Windows 95 with subsequent execution on AIX can make good use of resources when appropriate.

We use a traditional text editor and Java Servlet Development Kit (JSDK) tools to develop and test the servlet. The JSDK *srun* tool was useful for real-time feedback from the servlet. It prints output from the standard console `System.out.println()` and from servlet `log()` methods. The drawback, of course, is that *srun* is not a real Web server and cannot deliver the HTML or the applet. This is fine when using either VAJava or the JDK appletviewer tool.

The initial applet development was done totally inside VAJava. VAJava happily runs multiple instances of the applet, while allowing us to use all the facilities of its integrated development environment. Following the testing inside VAJava was the testing with appletviewer to bring in the HTML file with its parameters. Appletviewer also provides real-time feedback on applet operations using the `System.out.println()`.

We performed the last stages of testing with the target deployment environment, a Java 1.1-enabled Web browser (Netscape Communicator 4.03) for running the applet,

and a servlet-enabled Web server (Lotus Go 4.6.1) for running the servlet. Debugging can become somewhat more difficult in the real world because debugging capabilities digress to the 1970s methods of "printing" progress and values. For this reason, a few `System.out.println()` statements will appear in the applet. The results of such statements appear in the Netscape Navigator™ 4.0 Java Console, providing a simple way to track the applet progress when the browser just appears to "hang."

A few `log()` statements appear in the servlet where you might expect to find `System.out.println()`. Both work fine when running in the JSDK *srun* tool, but `System.out.println()` does nothing when running in a Web server environment. Web servers, however, keep logs of various activities, such as servlet activity, if configured to do so. The `log()` statement causes the argument string to show up in the Web server's servlet log, conveniently prefaced by a timestamp and the servlet name—a valuable aid for debugging and also useful for simple recording of servlet activity in the deployment environment.

Conclusion

Java has captured the imagination of the network computing industry. Much has been written about Java applets for clients, but servlets prove that "Java ain't just for clients anymore!" The tips and techniques presented here should make it easier for you to begin developing and debugging your own servlets—the next step in the evolution of network computing.

If you are interested in creating an environment similar to ours, we suggest you first compile and execute the application we've provided here. Sometimes it's easier to get a sample like ours running instead of creating a brand new program and trying to both compile and configure it on a new system. *Note:* See the Table of Contents for this issue of *AIXpert* for the complete Java code for this application.

Greg Flurry, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. Mr. Flurry is a senior technical staff member in the Server Development Division. His responsibilities, as part of the Systems Architecture & Technical Strategy team, include network computing and Java. He has a BS in Electrical Engineering from Vanderbilt University and an MS in the same field from the University of Kentucky.

Jeff Jilg, IBM Corporation, 11400 Burnet Road, Austin, TX 78758. E-mail: jjilg@austin.ibm.com. Dr. Jilg is a senior architect currently responsible for Server Development Java technical strategy in the Systems Architecture & Technical Strategy team.. His MS in Computer Science from the University of Texas at El Paso complements his PhD from Texas A&M University in the same field.