Building a Java chat server

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Tutorial tips 2
2. Introduction 3
3. First things first 5
4. The While-Accept loop (Server side) 8
5. Per-Thread class 10
6. The While-Read/Write loop (Server side) 12
7. Removing dead connections 14
8. The Client class 15
9. The While-Read/Write loop (Client side) 18
10. Limitations 19
11. Summary 21
12. Appendix: Source code listings 22

Section 1. Tutorial tips

Should I take this tutorial?

In this tutorial, we will build both the server and client sides of a simple chat system. This tutorial is for someone with little or no experience doing networking programming. We'll cover topics such as networking and multithreading in enough detail so that you'll be able to follow the examples, even if you have little or no experience doing this kind of programming. You will, however, need to be familiar with basic object-oriented programming in the Java language.

Navigation

Navigating through the tutorial is easy:

* Select Next and Previous to move forward and backward through the tutorial.
* When you're finished with a section, select the next section. You can also use the Main and Section Menus to navigate the tutorial.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Getting help

For questions about the content of this tutorial, contact the author, Greg Travis, at *mito@panix.com* .

Greg Travis is a freelance programmer living in New York City. His interest in computers can be traced back to that episode of "The Bionic Woman" where Jamie is trying to escape a building whose lights and doors are controlled by an evil artificial intelligence, which mocks her through loudspeakers. Greg is a firm believer that, when a computer program works, it's a complete coincidence.

Section 2. Introduction

What you'll learn

In this tutorial, you'll build a simple, centralized, connection-oriented Java server. In doing so, you'll learn a basic framework that you can use when creating such a server, using time-honored techniques that work well in many situations. We'll also examine some of the limitations of this framework and explore ways of getting around them.

What is a connection-oriented server?

Generally speaking, the job of any server is to provide a centralized service. However, there are many different ways of providing services, and many different ways to structure the communications.

Chat is roughly described as a connection-oriented service, because a user establishes a connection and maintains that connection, sending and receiving text for the duration of the session.

This is in contrast to the Web, where the protocol is (at least in theory) transactional -- the browser asks for a page, and the server sends it; the connection is then closed. (In practice, the connection is kept open and reused, but this is more a speed-optimization than a structuring metaphor.)

We'll be creating a stripped-down, connection-oriented server. Learning the basic framework will help you a great deal in creating other connection-oriented servers in the future.

Why create from scratch?

In creating this prototype server, we'll be using nothing more than the basic packages built into every Java implementation. This allows us to explore server programming at the very lowest level possible in the Java language.

There are certainly many systems available that can take care of many of these networking details for you. In many cases, the best real-world solution is to use an existing framework, because it often provides useful features such as fault-tolerance, load-balancing, and sessioning.

It is, nevertheless, crucial to understand how these things work at the lowest level. No existing solution is perfect for all problems, and existing solutions often have minor flaws that your code must work around. Simply choosing the right pre-packaged solution takes a discriminating eye that has been educated as to the tradeoffs inherent in various techniques.

Why a framework?

The word *framework* has a technical definition within the object-oriented community -- it

means a design structure that can be reused.

This is not the same as traditional code reuse, in which bits of code written for one purpose (or for no particular purpose at all) are reused for a new purpose. Rather, a framework is a reusable, overarching structure inside which you can implement your particular application.

One way to think of the difference between a framework and traditional code reuse is that traditional reuse involves inserting pre-existing elements into your particular structure, while a framework involves inserting your particular elements into a pre-existing framework.

The code that we'll develop in this tutorial isn't technically a framework, in that it is not structured to allow you to easily repurpose it for something else. Structuring it that way would distract from the purpose of this tutorial, which is to explore basic techniques of server programming.

However, in a design sense, it is a framework because the structure of the code is something you'll be able to use again and again.

Our framework

The framework we'll use in this tutorial has seven elements. Once you've gone through this tutorial and are familiar with these seven elements, you'll be able to use them again when you create your own connection-oriented server.

The seven elements are:

* Listener class
* While-Accept loop
* Per-Thread class
* While-Read/Write loop (Server side)
* Removing dead connections
* Client class
* While-Read/Write loop (Client side)

Section 3. First things first

What does the server do?

Before we describe the Listener class, we'll describe the server. Doing so has a certain chronological elegance, because in our running system, the server will have to start before any of the clients can connect to it.

Our server will be a stand-alone program -- a single Java process running on its own machine. It won't require any support software other than a Java virtual machine. And it won't require a Web server or application server, although a Web server or application server will likely be used to serve the client applet to the client.

More advanced server systems often embed the server code within a larger framework. This framework might be used to supply features such as load-balancing, special libraries for handling large numbers of clients, process migration, and database services. However, our example is going to stand all by itself. It will take care of all networking responsibilities on its own. As we'll see, this isn't very hard.

Listening on a port

The first thing we have to do is to get ready to receive incoming connections. To do this, we must listen on a port.

A port can be thought of as an address within a single computer. Remember that often a single machine might serve as a Web server, a chat server, an FTP server, and several other kinds of servers at the same time. Because of this, a connection to a server needs to specify not only the address of the machine itself, but also the particular service within the machine. This internal address is a port and is represented by a single integer between 1 and 65535.

Many standard services have a dedicated port number. For example, telnet uses port 23, FTP uses ports 20 and 21, and Web servers, by default, use port 80. Since our chat system is not famous (yet), we're going to have to use one of the port numbers allocated for general use.

We'll use port 5000. This means that our server is going to *listen* for connections on port 5000. Our clients, when connecting to our server machine, will specify that they want to connect to port 5000 on our server machine. This way, our clients and our server will be able to talk.

Sockets

Our communications between client and server will pass through a Java object called a Socket. Sockets are not at all Java-specific; the term is taken directly from the terminology of general IP (Internet Protocol) network programming. In Java programs, a Socket object is simply a wrapper around the low-level sockets that Internet programmers have been using for years. And the abstraction used by the Java language is very clean, so socket programming in the Java language is much more pleasant than it is, for example, in C.

The most important thing to know about a Socket object is that it contains (among other things) two Streams. One is for reading data coming in, and the other is for writing data out. That is to say, a Socket has an InputStream and an OutputStream.

(If these Stream classes are unfamiliar to you, then suffice it to say that they are objects used for reading and writing data, often as a stream of bytes. If you don't know about them yet, you really should. See the *java.io* package for more information.)

The class

So, now we get to the first of our seven elements, the *Listener Class*. We'll call it `Server.java`.

The next few panels will show the essential elements of this class: the constructor and the `main()` routine.

The constructor

The constructor for Server takes a single parameter -- a port number. This tells us what port to listen on when we are ready to start accepting connections. Here is the constructor:

```
// Constructor and while-accept loop all in one.
public Server( int port ) throws IOException {

  // All we have to do is listen
  listen( port );
}
```

Note that the `listen()` routine carries out the rest of the work of the server. We'll get to that in the next section.

The main() routine

We'll also include a `main()` routine so that this Server class can be used as its own stand-alone application. In practice, you might be embedding your basic server code in something larger, in which case you already have a `main()`. But for our purposes, the Server is all there is. Here's the `main()` routine:

```
// Main routine
// Usage: java Server >port<
static public void main( String args[] ) throws Exception {

  // Get the port # from the command line
  int port = Integer.parseInt( args[0] );

  // Create a Server object, which will automatically begin
  // accepting connections.
  new Server( port );
}
```

All set to listen

Now that we're all ready to listen, we'll continue to the next section to see how we accept new connections and what we do with them.

Section 4. The While-Accept loop (Server side)

The plan

We're ready to start accepting network connections from our clients. Here's how the chat is going to work.

We mentioned above that the Java language provides an object called a Socket, which represents a connection to a program somewhere else and through which data can pass.

But how do we get this socket in the first place? A client, almost by definition, initiates the connection to a server. Therefore, the first job of a server is to wait for a connection to come in. That is, we need something to give us the Sockets that are connected to our clients.

That's where the ServerSocket comes in. This is an object whose job is simple: listen on a port and when a new connection comes in, create a Socket that represents that new connection.

Accepting Sockets

Remember that your program will potentially be serving many clients from all over the Internet. And these clients will be connecting to your server without regard to each other. That is, there's no way to control the order, or the timing, with which the connections are arriving. As we'll see later, multithreading is an excellent way to deal with these connections once they have come in.

However, we're still trying to deal with the connections as they arrive.

The socket metaphor provides a straightforward solution: it *serializes* the incoming connections. That is, it makes them seem as if they are coming in one at a time, and arriving just as you ask for them.

Here's what it looks like, in the abstract:

```
// start listening on the port
ServerSocket ss = new ServerSocket( port );
// loop forever
while (true) {

  // get a connection
  Socket newSocket = ss.accept();

  // deal with the connection
  // ...
}
```

The `accept()` routine is the key here. When this method of ServerSocket is called, it returns a new Socket object that represents a new connection that has come in. After you've dealt with this connection, you call `accept()` and get the next one. This way, no matter how fast connections are coming, and no matter how many processors or network interfaces your machine has, you get the connections one at a time. (And if there aren't any connections coming in at the moment, then the `accept()` routine simply blocks -- doesn't return -- until there are.)

Serialization of incoming requests

Serialization is a useful way, in general, to deal with things that are happening simultaneously. A potential drawback, however, is that it can remove parallelism. That is to say, serialization can prevent us from saving time by working on multiple things at the same time. In the code above, while the program is dealing with one connection, other connections might be piling up.

But serialization is not a problem for us because each time a connection comes in, we're going to create a new thread to deal with it. Once the new thread is created, it can go off and deal with the new connection, and our while-accept loop can go back to accepting new connections. If the act of creating this new thread is fast enough, then the connections won't pile up.

The code

Let's take a look at the code that does all this. The code below takes care of the things we've been talking about: listening on a port, accepting new connections, and creating threads to deal with them. It also does a few other things that will be useful later. Let's take a look:

```
private void listen( int port ) throws IOException {

  // Create the ServerSocket
  ss = new ServerSocket( port );

  // Tell the world we're ready to go
  System.out.println( "Listening on "+ss );

  // Keep accepting connections forever
  while (true) {

    // Grab the next incoming connection
    Socket s = ss.accept();

    // Tell the world we've got it
    System.out.println( "Connection from "+s );

    // Create a DataOutputStream for writing data to the
    // other side
    DataOutputStream dout = new DataOutputStream( s.getOutputStream() );

    // Save this stream so we don't need to make it again
    outputStreams.put( s, dout );

    // Create a new thread for this connection, and then forget
    // about it
    new ServerThread( this, s );
  }
}
```

The last line of this listing creates a thread to deal with the new connection. This thread is an object called a ServerThread, which is the topic of the next section.

Section 5. Per-Thread class

Taking stock

Let's look at our list of seven elements again:

* Listener Class
* While-Accept Loop
* Per-Thread Class
* While-Read/Write Loop (Server)
* Removing Dead Connections
* Client Class
* While-Read/Write Loop (Client)

We've dealt with the first two, and in doing so, we've taken care of the core of the server. The next three steps are going to deal with the code that handles the connection to a particular client -- the Per-Thread class. We'll build this class in this section.

What is a thread?

Two of the Java language's main strengths are networking and multithreading. That is not to say that other languages don't support these functions -- they do. But the abstractions that the Java language uses to provide these features are particularly elegant, especially for a commercial language.

A thread is generally defined as a separate line of control within a single process. What this really means is that a multithreaded program has multiple, semi-autonomous activities going on inside of it at the same time.

Multithreading is similar to the concepts of a task and multitasking, except that the multiple threads within a program all share the same data space. This makes it easier for them to share data directly and efficiently -- and it also makes it easier for them to mess each other up.

Why use multithreading?

A detailed discussion of threads is beyond the scope of this tutorial. There are a few reasons why you'd want to use threads in your program, but there is one reason most pertinent to the construction of a chat server: input/output.

Your chat server is communicating (in a sense) with the users at the client. Users are usually much slower than servers, which means that your server code is going to spend a lot of time simply waiting for users to say things. And you never know who is going to say something first. If you have a single thread, and it's waiting for user #0 to say something, then it's not going to know that users #1 through #10 are talking like crazy.

For this reason, we're going to create a thread for *each* user connected to the system. The advantage of multithreading is that when one thread is listening for a slow user to say

something, it essentially goes to sleep until something comes in from that user. In the meantime, another thread can be receiving data from another user. In effect, multithreading allows us to respond as quickly as we can to each user.

All you really need to know about multithreading for this tutorial is that we're going to create a new thread for each connection. This thread, more or less by itself, is going to take care of that connection until it is severed.

The class

In Java programs, any class can be made into a thread by implementing the Runnable interface. Or you can simply subclass from the class `java.lang.Thread`. We have chosen the latter route, for no particular reason:

```
public class ServerThread extends Thread
{
  // ...
}
```

The constructor

Our constructor takes two parameters -- a Server object and a Socket object.

The Socket object is essential, because the whole purpose of this thread is to use a socket to communicate with the other side.

The Server object will come in handy later on.

Here's the code:

```
// Constructor.
public ServerThread( Server server, Socket socket ) {

  // Save the parameters
  this.server = server;
  this.socket = socket;

  // Start up the thread
  start();
}
```

Note that this constructor code is still being run in the main, server thread. The last line, where `start()` is called, is the point at which the new thread comes into being. This thread executes the `run()` method of this object, which we'll see in the next section.

Section 6. The While-Read/Write loop (Server side)

The communications protocol

Now that we've gotten to the point where we're going to be talking to the client, we should talk a little bit about our communication protocol.

Every client/server system has a communications protocol, which is nothing more than the format you use to send the data back and forth. The protocol can be so simple it hardly deserves the title of protocol, or it can be a sophisticated standard that has been ratified by consortia all over the world. Either way, it's a protocol.

We're going to create our own protocol, because in the Java language it's very easy to do, and because there's little for us to gain from using an existing standard. Our protocol will be very simple.

The Java language has a pair of extremely useful classes called DataInputStream and DataOutputStream. These classes allow you to read and write low-level data objects (like integers and strings) to a stream, without having to consider the format in which they are written. Because these classes use the same format, and because this format doesn't change, you can be sure that an integer written to a DataOutputStream will be properly read from the DataInputStream at the other end.

So our protocol will be this:

* When a user types something into their chat window, their message will be sent as a string through a DataOutputStream.
* When the server receives a message, through a DataInputStream, it will send this same message to all users, again as a string through a DataOutputStream.
* The users will use a DataInputStream to receive the message.

And that's it!

The loop

So now we get to our central server loop -- this is the code that does the real chat work. Let's take a look:

```
// This runs in a separate thread when start() is called in the
// constructor.
public void run() {

  try {

    // Create a DataInputStream for communication; the client
    // is using a DataOutputStream to write to us
    DataInputStream din = new DataInputStream( socket.getInputStream() );

    // Over and over, forever ...
    while (true) {

      // ... read the next message ...
      String message = din.readUTF();

      // ... tell the world ...
```

```
        System.out.println( "Sending "+message );

        // ... and have the server send it to all clients
        server.sendToAll( message );
      }
    } catch( EOFException ie ) {

      // This doesn't need an error message
    } catch( IOException ie ) {

      // This does; tell the world!
      ie.printStackTrace();
    } finally {

      // The connection is closed for one reason or another,
      // so have the server dealing with it
      server.removeConnection( socket );
    }
  }
```

You'll notice that we're now using our Server object in a couple of places.

The first place is when we call `server.sendToAll()`. This is really just a convenience function. We could just as well have asked the server for an Enumeration of all the OutputStreams connected to the other side, and then written the message to each of these streams. But because that is so frequently done, we moved the code for all of that work to the Server object, into a method called `sendToAll()`.

The second place we use our Server object is mentioned in the next section.

Section 7. Removing dead connections

Removing dead connections

The second place that our Per-Thread class refers to the main Server object is near the bottom, where we are removing a socket after the connection has closed. More precisely, our Per-Thread class, once it realizes that the connection has closed, must ask the Server to remove that connection so that we can be done with it. Here is the code:

```
// The connection is closed for one reason or another,
// so have the server dealing with it
server.removeConnection( socket );
```

It is important to inform the main Server object each time a connection has closed, so that the server can remove the connection from any internal lists, and also so that the server doesn't waste time sending messages to clients that are no longer connected.

The importance of cleaning up

This step of removing dead connections is given its own place as one of our seven elements of server construction because it's so crucial, and because it's often forgotten by people writing their first server.

If programmers forget this step, they find that their server works fine for a while. Then it begins to get slower and slower, and starts using more and more memory. When they look at the output of the server process, they see exceptions whizzing by at an alarming rate.

What's happened is that the server has kept around all the connections that have died, is trying to send messages to them, and is finding that it can't.

Let's say your server is receiving connections, on average, about once every five seconds. And let's further assume that the average user stays on about fifteen minutes.

This means you will have, on average, about 180 users on at any given moment. As time passes, many users will connect and disconnect, keeping the average around 180.

After 24 hours, 17,280 users will have connected to your server. At the end of this first day, you'll still have about 180 live connections, and you'll have about 17,100 dead connections. If you haven't taken care to remove these dead connections, then you can be sure that your server is spending most of its time trying, and failing, to write to these connections. (In fact, it might be spending most of its time spewing Exceptions to System.out!)

For this reason, it's *crucial* that you clean up after these dead connections.

Section 8. The Client class

The Client class

Believe it or not, we've finished building the server side of our chat system. We have one object (Server) listening for new connections and a bunch of per-connection objects (ServerThread) dealing with the connections themselves.

But what about the client side?

Our client is going be an applet, because we're assuming that one of the reasons you're using the Java language is that you want your program to be available on a Web page.

It doesn't have to be this way -- the client could just as easily be a stand-alone application, running in its own process just like the server is. Very little of the discussion below would be different if we went this route.

The Constructor: Set up the interface

Our client has a graphical user interface (GUI), because it must interact neatly with the user. The server could get by with just a command-line interface, because once it starts running, the user (or administrator) doesn't really need to do anything. But the whole point of the client side is user interaction.

The first thing our client is going to do is set up the interface:

```
// Constructor
public Client( String host, int port ) {

  // Set up the screen
  setLayout( new BorderLayout() );
  add( "North", tf );
  add( "Center", ta );

  // We want to receive messages when someone types a line
  // and hits return, using an anonymous class as
  // a callback
  tf.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      processMessage( e.getActionCommand() );
    }
  } );

// ...
```

We won't go into too much detail about this, except to say that our chat window contains a text entry field, for typing new messages, and a text display window, for showing messages from other users. Each time the user types something into the input field, the string is passed to the method `processMessage()`.

The Constructor: Connect to the server

The next thing the constructor does is initiate a connection to the server, as follows:

```
    // Connect to the server
    try {

      // Initiate the connection
      socket = new Socket( host, port );

      // We got a connection!  Tell the world
      System.out.println( "connected to "+socket );

      // Let's grab the streams and create DataInput/Output streams
      // from them
      din = new DataInputStream( socket.getInputStream() );
      dout = new DataOutputStream( socket.getOutputStream() );

      // Start a background thread for receiving messages
      new Thread( this ).start();
    } catch( IOException ie ) { System.out.println( ie ); }
  }
```

Note that we've created a separate thread to process incoming messages. We'll do this in the next section.

User input

An applet runs as a component embedded in a larger GUI framework. In this framework, code generally runs in response to an input event received in one of the GUI windows of the program. The applet processes the event, doing what it needs to do, and then it returns, waiting for the system to send another event along.

In this case, the user might type something in the input field. This triggers a call to the anonymous inner class we created in the constructor. This anonymous inner class, in turn, calls `processMessage()`, which is a method we've created.

This method is passed the string that the user typed. What this method then does is simple: it writes the string to the server, and then clears out the text input field so that the user can type something else. Here is the code:

```
  // Gets called when the user types something
  private void processMessage( String message ) {
    try {

      // Send it to the server
      dout.writeUTF( message );

      // Clear out text input field
      tf.setText( "" );
    } catch( IOException ie ) { System.out.println( ie ); }
  }
```

Background thread

Our program isn't just waiting for GUI events. It's also waiting for network events. That is, it's waiting for data to come from the server. And since we don't already have a built-in framework for receiving these messages through callbacks, we have to set something up ourselves.

Remember that the last thing our constructor did was to create a new thread, a little bit like

the server does. However, this is done for a very different reason, which is that we are running as an applet.

In this case, we create a background thread that will run the client's while-read/write loop. We'll do this in the next section.

Section 9. The While-Read/Write loop (Client side)

The While-Read/Write loop (Client side)

Just as we have a read/write loop on the server, we have one on the client. The purpose is the same: read an incoming message, process it, and maybe write some stuff back to the server in response. Then do it all over again.

Here's what our client-side loop looks like:

```
// Background thread runs this: show messages from other window
public void run() {
  try {

    // Receive messages one-by-one, forever
    while (true) {

      // Get the next message
      String message = din.readUTF();

      // Print it to our text window
      ta.append( message+"\n" );
    }
  } catch( IOException ie ) { System.out.println( ie ); }
}
```

Pretty simple. Each incoming message gets displayed in the text display window, and then the loop goes back to waiting for the next message.

Section 10. Limitations

Limitations

At this point, we have a complete, albeit stripped-down, multithreaded chat system. What could possibly be wrong?

As was mentioned in the introduction, no server framework is right for all tasks. It's important to know the limitations of this particular method, so that you can decide whether or not you should apply these ideas to your own projects.

Simplistic connection model

The chat system we've created has only one chat room. Remember that every message that came into the server was sent out to every single client. This is not satisfactory in the real world, not only because real-world users want separate chat rooms, but because it's a terrible waste of bandwidth.

Of course, there's nothing in our model that prevents you from allowing users to select a chat room, and then associating each message with a room name, so that messages are only sent out to users in the same room as the sender.

However, we didn't leave a place for this selection. Our ServerThread object simply calls a method in our Server object called `sendToAll()`. In the real world, the ServerThread is going to have to forward the message to a real dispatch system.

Too many threads

In some Java implementations, it's not a good idea to create a thread for each client, because having a lot of threads -- even if they are all asleep -- can bog down the system.

This is something you'll have to test on your own, because there's no easy way to predict threading performance without trying it. And if it *does* turn out that your Java implementation can't handle it, the solution can be rather hairy.

In this case, you'll have to use a small, but bounded number of threads to read from the sockets, and you'll have to use non-blocking threads to do so, which means you'll have to use polling, which means ... and so on. It's a whole can of worms. If your JVM can't handle a lot of threads, call your vendor and complain!

Internal synchronization

The comments inside the code for `Server.java` mention that some thread synchronization is used to prevent problems related to properly maintaining the list of active connections. This is just the kind of synchronization that can really impact performance.

If synchronization turns out to be a bottleneck, there are a couple of things you can try:

* Change `removeConnection()` to simply make a note of which connection should be removed, perhaps in another list. Then, do the actual removing inside `sendToAll()`, during or after traversal.
* Use more than one thread to do the writing in `sendToAll()`.

Section 11. Summary

What you've learned

This tutorial has gone into great detail about the ideas used in the construction of a multithreaded, connection-oriented server in the Java language. You've learned how to deal with multiple clients at once, and you've come to know the seven basic elements that make up such a server.

Resources

These books provide valuable information on Java programming:

* *"Java Network Programming, 2nd Edition"* by Elliotte Rusty Harold
* *"Java Network Programming, 2nd Edition"* by Merlin Hughes, Michael Shoffner, and Derek Hamner (different book with the same title)
* *"Java 1.1 Unleashed"*

Other tutorials related to this topic are:

* *Introduction to JavaServer Pages technology*
* *Building Java HTTP servlets*
* *Building servlets with session tracking*

The *Java Network Programming FAQ* has answers to many questions.

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author, Greg Travis, at *mito@panix.com* .

Section 12. Appendix: Source code listings

Server.java

```
// $Id$

import java.io.*;
import java.net.*;
import java.util.*;

public class Server
{
  // The ServerSocket we'll use for accepting new connections
  private ServerSocket ss;

  // A mapping from sockets to DataOutputStreams.  This will
  // help us avoid having to create a DataOutputStream each time
  // we want to write to a stream.
  private Hashtable outputStreams = new Hashtable();

  // Constructor and while-accept loop all in one.
  public Server( int port ) throws IOException {

    // All we have to do is listen
    listen( port );
  }

  private void listen( int port ) throws IOException {

    // Create the ServerSocket
    ss = new ServerSocket( port );

    // Tell the world we're ready to go
    System.out.println( "Listening on "+ss );

    // Keep accepting connections forever
    while (true) {

      // Grab the next incoming connection
      Socket s = ss.accept();

      // Tell the world we've got it
      System.out.println( "Connection from "+s );

      // Create a DataOutputStream for writing data to the
      // other side
      DataOutputStream dout = new DataOutputStream( s.getOutputStream() );

      // Save this stream so we don't need to make it again
      outputStreams.put( s, dout );

      // Create a new thread for this connection, and then forget
      // about it
      new ServerThread( this, s );
    }
  }

  // Get an enumeration of all the OutputStreams, one for each client
  // connected to us
  Enumeration getOutputStreams() {
    return outputStreams.elements();
  }

  // Send a message to all clients (utility routine)
  void sendToAll( String message ) {

    // We synchronize on this because another thread might be
    // calling removeConnection() and this would screw us up
    // as we tried to walk through the list
    synchronized( outputStreams ) {

      // For each client ...
      for (Enumeration e = getOutputStreams(); e.hasMoreElements(); ) {
```

```
        // ... get the output stream ...
        DataOutputStream dout = (DataOutputStream)e.nextElement();

        // ... and send the message
        try {
          dout.writeUTF( message );
        } catch( IOException ie ) { System.out.println( ie ); }
      }
    }
  }

  // Remove a socket, and it's corresponding output stream, from our
  // list.  This is usually called by a connection thread that has
  // discovered that the connectin to the client is dead.
  void removeConnection( Socket s ) {

    // Synchronize so we don't mess up sendToAll() while it walks
    // down the list of all output streamsa
    synchronized( outputStreams ) {

      // Tell the world
      System.out.println( "Removing connection to "+s );

      // Remove it from our hashtable/list
      outputStreams.remove( s );

      // Make sure it's closed
      try {
        s.close();
      } catch( IOException ie ) {
        System.out.println( "Error closing "+s );
        ie.printStackTrace();
      }
    }
  }

  // Main routine
  // Usage: java Server <port>
  static public void main( String args[] ) throws Exception {

    // Get the port # from the command line
    int port = Integer.parseInt( args[0] );

    // Create a Server object, which will automatically begin
    // accepting connections.
    new Server( port );
  }
}
```

ServerThread.java

```
// $Id$

import java.io.*;
import java.net.*;

public class ServerThread extends Thread
{
  // The Server that spawned us
  private Server server;

  // The Socket connected to our client
  private Socket socket;

  // Constructor.
  public ServerThread( Server server, Socket socket ) {

    // Save the parameters
    this.server = server;
```

```
    this.socket = socket;

    // Start up the thread
    start();
  }

  // This runs in a separate thread when start() is called in the
  // constructor.
  public void run() {

    try {

      // Create a DataInputStream for communication; the client
      // is using a DataOutputStream to write to us
      DataInputStream din = new DataInputStream( socket.getInputStream() );

      // Over and over, forever ...
      while (true) {

        // ... read the next message ...
        String message = din.readUTF();

        // ... tell the world ...
        System.out.println( "Sending "+message );

        // ... and have the server send it to all clients
        server.sendToAll( message );
      }
    } catch( EOFException ie ) {

      // This doesn't need an error message
    } catch( IOException ie ) {

      // This does; tell the world!
      ie.printStackTrace();
    } finally {

      // The connection is closed for one reason or another,
      // so have the server dealing with it
      server.removeConnection( socket );
    }
  }
}
```

Client.java

```
// $Id$

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class Client extends Panel implements Runnable
{
  // Components for the visual display of the chat windows
  private TextField tf = new TextField();
  private TextArea ta = new TextArea();

  // The socket connecting us to the server
  private Socket socket;

  // The streams we communicate to the server; these come
  // from the socket
  private DataOutputStream dout;
  private DataInputStream din;

  // Constructor
  public Client( String host, int port ) {
```

```
    // Set up the screen
    setLayout( new BorderLayout() );
    add( "North", tf );
    add( "Center", ta );

    // We want to receive messages when someone types a line
    // and hits return, using an anonymous class as
    // a callback
    tf.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent e ) {
        processMessage( e.getActionCommand() );
      }
    } );

    // Connect to the server
    try {

      // Initiate the connection
      socket = new Socket( host, port );

      // We got a connection!  Tell the world
      System.out.println( "connected to "+socket );

      // Let's grab the streams and create DataInput/Output streams
      // from them
      din = new DataInputStream( socket.getInputStream() );
      dout = new DataOutputStream( socket.getOutputStream() );

      // Start a background thread for receiving messages
      new Thread( this ).start();
    } catch( IOException ie ) { System.out.println( ie ); }
  }

  // Gets called when the user types something
  private void processMessage( String message ) {
    try {

      // Send it to the server
      dout.writeUTF( message );

      // Clear out text input field
      tf.setText( "" );
    } catch( IOException ie ) { System.out.println( ie ); }
  }

  // Background thread runs this: show messages from other window
  public void run() {
    try {

      // Receive messages one-by-one, forever
      while (true) {

        // Get the next message
        String message = din.readUTF();

        // Print it to our text window
        ta.append( message+"\n" );
      }
    } catch( IOException ie ) { System.out.println( ie ); }
  }
}
```

ClientApplet.java

```
// $Id$

import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;
```

```
public class ClientApplet extends Applet
{
  public void init() {
    String host = getParameter( "host" );
    int port = Integer.parseInt( getParameter( "port" ) );
    setLayout( new BorderLayout() );
    add( "Center", new Client( host, port ) );
  }
}
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.