Chapter 17

Networking in Java

- Everything You Need To Know About TCP/IP but Failed to Learn in Kindergarten
- A Client Socket in Java
- Sending Email by Java
- A Server Socket in Java
- HTTP and Web Browsing: Retrieving HTTP Pages
- How to Make an Applet Write a File on the Server
- A Multithreaded HTTP Server
- A Mapped I/O HTTP Server
- Further Reading
- Exercises
- Some Light Relief—Using Java to Stuff an Online Poll

“If a packet hits a pocket on a socket on a port, and the bus is interrupted and the interrupt’s not caught, then the socket packet pocket has an error to report.”

— Programmer’s traditional nursery rhyme

The biggest barrier to understanding Java networking features is getting familiar with network terms and techniques. If you speak French, it doesn’t mean that you can understand an article from a French medical journal.

Similarly, when you learn Java, you also need to have an understanding of the network services and terminology before you can write Internet code. So this chapter starts with the basics of TCP/IP networking, followed by a description of Java support.

There is a lot of knowledge in this chapter. After the TCP/IP basics, we’ll develop some socket examples. We’ll see how a client gets services from a remote server using sockets. Then we will look at server sockets to see how incoming connections are accepted. Our first example will merely print HTTP headers. We will add to it little by little until it is a complete multithreaded HTTP web server. Finally, we will show how to use new I/O channels with server sockets.
Networking at heart is all about shifting bits from point A to point B. We bundle the data bits into a packet, and add some more bits to say where they are to go. That, in a nutshell, is the Internet Protocol or IP. If we want to send more bits than will fit into a single packet, we can divide the bits into groups and send them in several successive packets. The units that we send are called “User Datagrams” or “Packets.” Packets is the more common term these days.

User Datagrams can be sent across the Internet using the User Datagram Protocol (UDP), which relies on the Internet Protocol for addressing and routing. UDP is like going to the post office, sticking on a stamp, and dropping off the packet. IP is what the mail carrier does to route and deliver the packet. Two common applications that use the UDP are: SNMP, the Simple Network Management Protocol, and TFTP, the Trivial File Transfer Protocol. See Figure 17-1.

When we send several pieces of postal mail to the same address, the packages might arrive in any order. Some of them might even be delayed, or even on occasion lost altogether. This is true for UDP too; you wave goodbye to the bits as they leave your workstation, and you have no idea when they will arrive where you sent them, or even if they did.

Uncertain delivery is equally undesirable for postal mail and for network bit streams. We deal with the problem in the postal mail world (when the importance warrants the cost) by paying an extra fee to register the mail and have the mail carrier collect and bring back a signature acknowledging delivery. A similar protocol is used in the network work to guarantee reliable delivery in the order in which the packets were sent. This protocol is known as Transmission Control Protocol or “TCP.” Two applications that run on top of, or use, TCP are: FTP, the File Transfer Protocol, and Telnet.

What Is Your IP Address?

On Unix workstations, you can run the “ifconfig” (interface configuration) program to find out your IP address.

On Windows 9x, you can run WinIPCfg to get the same information. Type this in a command tool:

c:\> winipcfg

It will popup a window that lists the host name, IP address, subnet mask, gateway, and even the MAC address of your network card.

The MAC (Media Access Control) address is the address on the network interface card burned in at manufacturing time. It is not used in TCP/IP because, unlike IP addresses, it lacks a hierarchy. To route packets using MAC addresses, each router would need a list of every MAC address in the world.
System A has a big block of information to send to system B.

The IP software chops it into manageable sized packets, sticks an address on the front, and shoots those user datagrams over to system B.

But the datagrams might not arrive, or might arrive out of order.

That's B's problem!

Figure 17–1 IP and UDP (datagram sockets).
TCP uses IP as its underlying protocol (just as UDP does) for routing and delivering the bits to the correct address. The “correct address” means the IP address; every computer on the Internet has an IP address. However, TCP is more like a phone call than a registered mail delivery in that it supports an end-to-end connection for the duration of the transmission session. It takes a while to set up this stream connection, and it costs more to assure reliable sequenced delivery, but the cost is usually justified. See Figure 17-2.

The access device at each endpoint of a phone conversation is a telephone. The access object at each endpoint of a TCP/IP session is a socket. Sockets started life as a way for two processes on the same Unix system to talk to each other, but some smart programmers realized that they could be generalized into connection endpoints between processes on different machines connected by a TCP/IP network. Today, every operating system has adopted IP and sockets.

IP can deliver the following via socket connections:

- Slower reliable delivery using TCP (this is termed a stream socket)
- Faster but unguaranteed delivery using UDP (this is a datagram socket)
- Fast raw bits using ICMP (Internet Control Message Protocol) datagrams. They are not delivered at all, but ask the remote end to do something or respond in some way.

ICMP is a low-level protocol for message control and error reporting. It uses IP packets, but its messages are directed at the IP software itself and don’t come through to the application layer. Java doesn’t support ICMP and we won’t say anything more about it.

Socket connections have a client end and a server end, and they differ in what you can do with them. Generally, the server end just keeps listening for incoming requests (an “operators are standing by” kind of thing). The client end initiates a connection, and then passes or requests information from the server.

Note that the number of socket writes is not at all synchronized with the number or timing of socket reads. A packet may be broken into smaller packets as it is sent across the network, so your code should never assume that a read will get the same number of bytes that were just written into the socket.

The most widely used version of IP today is Internet Protocol Version 4 (IPv4). However, IP Version 6 (IPv6 or IPng) is also beginning to enter the market. IPv6 uses 128 bit addresses, not 32 bit, and so allows many more Internet users. IPv6 is fully backward compatible with (can process packets sent using) IPv4, but it will take a long time before IPv4 is displaced by v6. IPv4 is supported with hardware-based routing at wire speed on 2.5Gb links. IPv6 currently uses software routing.

An IPv4 feature called “Network Address Translation” (NAT) has reduced the pressure to move to v6. A few years ago, it looked like we were going to run
TCP/IP (stream sockets)

System A has a big block of information to send to system B. The data must be sent reliably.

The IP software chops it into manageable sized packets, sticks an address on the front, and sends those packets to system B.

The internet path between the two sockets is held open until all the data is sent.

The packets are guaranteed to arrive, and are put in order as missed packets are resent.

Figure 17–2 TCP/IP (stream sockets).
Just Java 2

out of IP addresses. Today NAT lets your big site have just one assigned address, which you use for the computer with the internet connection. You use any IP address you like for the computers on your side of the firewall. You may be duplicating numbers that someone else uses behind their firewall, but the two systems don’t interfere with each other. When you access the internet, NATS translates your internal IP address into the externally visible one, and vice versa for incoming packets. From outside, it looks like all your traffic is coming from your computer that runs NATS.

Looking at a Packet Traveling over the Net

Packets are moved along by routers, which are special-purpose computers that connect networks. Every IP packet that leaves your system goes to a nearby router which will move the packet to another router closer to the destination. This transfer continues until finally the packet is brought to a router that is directly connected to the subnet serving the destination computer.

Routers maintain large configuration tables of what addresses are served by what routers, what the priorities are, and what rules they should use for security and load balancing. These tables can be updated dynamically as the network runs.

Windows has a program that lets you trace a packet’s movement between routers. Here’s the output from a sample run, tracing the route between my PC and java.sun.com. Unix has a similar program, called “traceroute.”

c:\> tracert java.sun.com
Tracing route to java.sun.com [192.18.97.71] over a maximum of 30 hops:
1    93 ms    95 ms    95 ms  sdn-ar-008carcor001t.dialsprint.net
     [63.128.147.130]
2    94 ms 100 ms 100 ms  sdn-hr-008carcor001t.dialsprint.net
     [63.128.147.129]
3    99 ms 100 ms  95 ms  sdn-pnc1-stk-4-1.dialsprint.net
     [207.153.212.49]
... and so on to ...
12   164 ms   170 ms  160 ms  sun-1.border3.den.pnap.net
     [216.52.42.42]
13  166 ms 160 ms  161 ms  java.sun.com [192.18.97.71]
Trace complete.

This shows that it takes 13 “hops” for packets to travel from my PC to Sun’s Java website. The program sends three test packets and notes the round trip time in milliseconds to reach each successive router. It works by sending out packets with brief time limits, and gradually increasing it until the first router gets it, and then the next, and so on. As each router replies, objecting to the timed-out packet, traceroute can figure out the hop time for each step. Traceroute is good for determining network connectivity.

Here it tells us that overall packets travel from me to Java-HQ in under a fifth of a second.
There! Now you know everything you need to use the Java networking features.

**What's in the Networking Library?**

If you browse the network library API, you’ll find the following classes (there are a few other classes, but these are the key ones):

- **Socket**
  This is the client Socket class. It lets you open a connection to another machine, anywhere on the Internet (that you have permission).

- **ServerSocket**
  This is the server Socket class. ServerSocket lets an application accept TCP connections from other systems and exchange I/O with them.

- **URL**
  The class represents a Uniform Resource Locator—a reference to an object on the web. You can create a URL reference with this class.

- **URLConnection**
  You can open a URL and retrieve the contents, or write to it, using this class.

- **HttpURLConnection**
  The class extends URLConnection and supports functions specific to HTTP, like get, post, put, head, trace, and options.

- **URLEncoder/URLDecoder**
  These two classes have static methods to allow you to convert a String to and from MIME x-www-form-urlencoded form. This is convenient for posting data to servlets or CGI scripts.

The class DatagramSocket supports the use of UDP packets. We don’t deal with UDP here because it is much less widely used than TCP. Most people want the reliability feature that TCP offers. Ironically, the widespread use of subnets using directly connected switches (instead of shared ethernet) has made UDP much more reliable, to the point where people are using it on LANs instead of TCP, and getting performance and reliability.

Let me try that last sentence again. When we started extensive networking in the late 1970s, ethernet was the medium of choice. You strung a single ethernet cable down a corridor and workstations physically attached to the net by tapping into the cable. That meant that all the network traffic was visible to all the work-
stations that used that cable. It was electronically noisy and slow. Today, nearly everyone uses 10baseT or 100baseT wiring. The number is the speed in Megabits, and the “T” part means “Twisted pair.” There is a twisted pair wire from your workstation directly to the switch that controls your subnet. No other workstation shares your twisted pair wiring. Result: faster performance, less electronic noise, and more reliable subnets, leading to greater confidence using UDP.

**TCP/IP Client/Server Model**

Before we look at actual Java code, a diagram is in order showing how a client and server typically communicate over a TCP/IP network connection. Figure 17-3 shows the way the processes contact each other is by knowing the IP address (which identifies a unique computer on the Internet) and a port number (which is a simple software convention the OS maintains, allowing an incoming network connection to be directed to a specific process).

---

**What Is a Socket?**

A socket is defined as “an IP address plus a port on that computer.”

---

![Diagram of client and server communication using a TCP/IP connection.](image-url)

**Figure 17–3** Client and server communication using a TCP/IP connection.
An IP address is like a telephone number, and a port number is like an extension at that number. Together they specify a unique destination. As a matter of fact, a socket is defined as an IP address and a port number.

The client and server must agree on the same port number. The port numbers under 1024 are reserved for system software use and on Unix can only be accessed by the superuser.

For simplicity, network socket connections are made to look like I/O streams. You simply read and write data using the usual stream methods (all socket communication is in 8-bit bytes), and it automagically appears at the other end. Unlike a stream, a socket supports two-way communication. There is a method to get the input stream of a socket, and another method to get the output stream. This allows the client and server to talk back and forth.

Almost all Internet programs work as client/server pairs. The server is on a host system somewhere in cyberspace, and the client is a program running on your local system. When the client wants an Internet service (such as retrieving a web page from an HTTP server), it issues a request, usually to a symbolic address such as www.sun.com rather than to an IP address (though that works, too).

There will be a Domain Name Server locally (usually one per subnet, per campus, or per company) that resolves the symbolic name into an Internet address.

The bits forming the request are assembled into a datagram and routed to the server. The server reads the incoming packets, notes what the request is, where it came from, and then tries to respond to it by providing either the service (web page, shell account, file contents, etc.) or a sensible error message. The response is sent back across the Internet to the client.

All the standard Internet utilities (telnet, rdist, FTP, ping, rcp, NFS, and so on) operate in client/server mode connected by a TCP or UDP socket. Programs that send mail don’t really know how to send mail—they just know how to take it to the Post Office. In this case, mail has a socket connection and talks to a demon at the other end with a fairly simple protocol. The standard mail demon knows how to accept text and addresses from clients and transmit it for delivery. If you can talk to the mail demon, you can send mail. There is little else to it.

Many of the Internet services are actually quite simple. But often considerable frustration comes in doing the socket programming in C and in learning the correct protocol. The socket programming API presented to C is quite low-level and all too easy to screw up. Needless to say, errors are poorly handled and diagnosed. As a result, many programmers naturally conclude that sockets are brittle and hard to use. Sockets aren’t hard to use. The C socket API is hard to use.

The C code to establish a socket connection is:
int set_up_socket(u_short port) {
    char   myname[MAXHOSTNAME+1];
    int    s;
    struct sockaddr_in sa;
    struct hostent *he;

    bzero(&sa,sizeof(struct sockaddr_in)); /* clear the address */
    gethostname(myname,MAXHOSTNAME);      /* establish identity */
    he= gethostbyname(myname);            /* get our address */
    if (he == NULL) /* if addr not found... */
        return(-1);
    sa.sin_family= he->h_addrtype;        /* host address */
    sa.sin_port= htons(port);             /* port number */

    if ((s= socket(AF_INET,SOCK_STREAM,0)) <0) /* finally, create socket */
        return(-1);
    if (bind(s, &sa, sizeof(sa), 0) < 0) {
        close(s);
        return(-1); /* bind address to socket */
    }

    listen(s, 3);                        /* max queued connections */
    return(s);
}

By way of contrast, the equivalent Java code is:

ServerSocket servsock = new ServerSocket(port, 3);

That's it! Just one line of Java code to do all the things the C code does.

Java handles all that socket complexity “under the covers” for you. It doesn’t expose the full range of socket possibilities, so Java avoids the novice socketeer choosing contradictory options. On the other hand, a few recondite sockety things cannot be done in Java. You cannot create a raw socket in Java, and hence cannot write a ping program that relies on raw sockets (you can do something just as good though). The benefit is overwhelming: You can open sockets and start writing to another system just as easily as you open a file and start writing to hard disk.

A “ping program,” in case you’re wondering, is a program that sends ICMP control packets over to another machine anywhere on the Internet. This action is called “pinging” the remote system, rather like the sonar in a ship “pings” for submarines or schools of fish. The control packets aren’t passed up to the application layer, but tell the TCP/IP library at the remote end to send back a reply. The reply lets the pinger calculate how quickly data can pass between the two systems.
The Story of Ping
If you want to know how quickly your packets can reach a system, use ping.

c:\> ping java.sun.com
Pinging java.sun.com [192.18.97.71] with 32 bytes of data:
Reply from 192.18.97.71: bytes=32 time=163ms TTL=241
Ping statistics for 192.18.97.71:
   Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
   Approximate round trip times in milli-seconds:
      Minimum = 160ms, Maximum = 169ms, Average = 163ms

This confirms that the time for a packet to hustle over from Mountain View to Cupertino is about 0.16 seconds on this particular day and time. “TTL” is “Time to Live.” To prevent infinite loops, each router hop decrements this field in a packet, and if it reaches zero, the packet just expires where it is.

The most used methods in the API for the client end of a socket are:

```java
public class Socket extends Object {
    public Socket();
    public Socket(String, int) throws UnknownHostException,
        java.io.IOException;
    public Socket(InetAddress, int) throws java.io.IOException;

    public java.nio.channels.SocketChannel getChannel();
    public InputStream getInputStream() throws IOException;
    public OutputStream getOutputStream() throws IOException;
    public synchronized void setSoTimeout(int) throws SocketException;
    public synchronized void close() throws IOException;
    public boolean isConnected();
    public boolean isBound();
    public boolean isClosed();
    public boolean isInputShutdown();
    public boolean isOutputShutdown();
    public boolean shutdownOutput() throws IOException;
    public boolean shutdownInput() throws IOException;
    public static void setSocketImplFactory(
        SocketImplFactory fac);
}
```
The constructor with no arguments creates an unconnected socket which you can later \texttt{bind()} to a host and port you specify. After binding, you will \texttt{connect()} it. It’s easier just to do all this by specifying these arguments in the constructor, if you know them at that point.

The \texttt{setSoTimeout(int ms)} will set a timeout on the socket of ms milliseconds. When this is a non-zero amount, a read call on the input stream will block for only this amount of time. Then it will break out of it by throwing a \texttt{java.net.SocketTimeoutException}, but leaving the socket still valid for further use.

The \texttt{setSocketFactory()} method is a hook for those sites that want to provide their own implementation of sockets, usually to deal with firewall or proxy issues. If this is done, it will be done on a site-wide basis, and individual programmers won’t have to worry about it.

The socket API has one or two dozen other get/set methods for TCP socket options. Most of the time you don’t need these and can ignore them.
A Client Socket in Java

This section shows a simple example of using a socket to communicate with another computer. You should type this code in and try it. If you haven’t done much network programming, you’ll find it a gleeful experience as you network with systems around the planet, and even in space. The space shuttle has a TCP/IP connection to Mission Control, but the spoilers at NASA keep its address secret.

There is an Internet protocol known as Network Time Protocol or NTP. NTP is used to synchronize the clocks of some computers. Without periodic syncing, computer clocks tend to drift out of alignment, causing problems for times they need to agree on, like email and file timestamps. NTP is pretty fancy these days, but a simple part of the protocol involves making a socket connection to a NTP server to get the time.

Our example program will open a socket connection to an NTP server and print out the time it gets back. The way a client asks for the time is simply to make a socket connection to port 13 on an NTP server. Port 13 is the Internet standard on all computers for the time of day port. You don’t have to identify yourself or write some data indicating what you want. Just making the socket connection is enough to get the server to give you an answer. Java does all the work of assembling the bytes into packets, sending them, and giving you an input stream with the bytes coming back from the server.

Here is a Java program that connects to an NTP server and asks the time:

```java
import java.io.*;
import java.net.*;
public class AskTime {
    public static void main(String a[]) throws Exception {
        if (a.length!=1) {
            System.out.println("usage: java AskTime <systemname> ");
            System.exit(0);
        }
        String machine = a[0];
        final int daytimeport = 13;
        Socket so = new Socket(machine, daytimeport);
        BufferedReader br =
            new BufferedReader(new InputStreamReader(
                so.getInputStream()));
        String timestamp = br.readLine();
        System.out.println( machine + " says it is " + timestamp );
    }
}
```
The program expects the name of an NTP server to be passed to it on the commandline. There are about 200,000 NTP servers on the Internet. Several national standards organizations allow reading the time via NTP. Table 17-1 gives some addresses for the service.

Table 17–1  Some Global Timeservers

<table>
<thead>
<tr>
<th>Organization</th>
<th>NTP Server</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physikalisch-technischen Bundesanstalt, Germany</td>
<td>ptbtime1.ptb.de</td>
<td>194.95.250.35</td>
</tr>
<tr>
<td>Mass Inst. Technology, USA</td>
<td>tick.mit.edu</td>
<td>18.145.0.30</td>
</tr>
<tr>
<td>US Naval Observatory, Washington, DC</td>
<td>tock.usno.navy.mil</td>
<td>192.5.41.41</td>
</tr>
<tr>
<td>Univ. of Adelaide, Australia</td>
<td>ntp.adelaide.edu.au</td>
<td>129.127.40.3</td>
</tr>
<tr>
<td>Inet Inc., Seoul, Korea</td>
<td>time.nuri.net</td>
<td>203.255.12.4</td>
</tr>
</tbody>
</table>

These servers come and go. Do a web search on "NTP server" for a current list. When you run the program, giving a hostname as argument, you see this:

```bash
% java AskTime ntp.adelaide.edu.au
```
You can even provide the IP address instead of the server name, and the program will work equally well.

This program demonstrates how easy it is to open a socket connection to a port on another computer using the Java networking library. It’s just flat out impressive to write a dozen lines of code that can ask a computer anywhere on the planet to tell you the time. Maybe there’s something to this Internet thing, after all.

Sockets are used in client or in server mode. The program above is an example of the client use of socket. The client side initiates the contact. It is like knocking on a door or calling a phone number and starting a conversation with whoever answers.

The server side is just sitting there, waiting on a socket until someone shows up to ask for something. We will show how to write a server socket a little later in the chapter. The next topic is another example of how a client can obtain a service by opening a socket connection and writing to it. The example here is sending email, by writing to the mailserver port which (as another Internet standard) lives on port 25.
Sending Email by Java

As our next example, let’s write a Java program to send some email. Email is sent by socketed communication with port 25 on a computer system. All we are going to do is open a socket connected to port 25 on some system that is running a mail server and speak “mail protocol” to the sendmail demon at the other end. If we speak the mail protocol correctly, it will listen to what we say, and send the email for us.

The following below requires an Internet standard mail (SMTP) program running on the server. If your server has some non-standard proprietary mail program on it, you’re out of luck. You can check which program you have by telnetting to port 25 on the server, and seeing if you get a mail server to talk to you.

There are two wrinkles to this approach. First, it became common for spammers to steal time on other people’s mailservers to relay their spam. As a result, most mail servers are now selective about who they accept a connection from. You won’t be able to get mailers around the world to talk to you, just your ISP mail server. Second, Java now has a mail API with a somewhat higher-level interface, so you don’t need to understand individual mail commands. But the point here is to show some give and take over a socket connection. Again, this example shows the client end of the socket connection.

The code to send email is:
import java.io.*;
import java.net.*;

public class email {
    public static void main(String args[]) throws IOException {
        Socket sock;
        DataInputStream dis;
        PrintStream ps;

        sock = new Socket("localhost", 25);
        dis = new DataInputStream( sock.getInputStream());
        ps = new PrintStream( sock.getOutputStream());

        ps.println("mail from: trelford");
        System.out.println( dis.readLine() );

        String Addressee= "linden";
        ps.println("rcpt to: " + Addressee );
        System.out.println( dis.readLine() );

        ps.println("data");
        System.out.println( dis.readLine() );

        ps.println("This is the message\n that Java sent");
        ps.println(".");
        System.out.println( dis.readLine() );

        ps.flush();
        sock.close();
    }
}
Just Java 2

Running this program will send email to the addressee. Many of the Internet services are like this one. You set up a socket connection and talk a simple protocol to tell the server at the other end what you want.

Note that the main() routine has been declared as throwing an Exception. This is a shortcut, permissible in development, to save the necessity of handling any exceptions that might be raised. It only works because exceptions are not considered part of the signature of a method. In production code, it is important to catch and handle any exceptions.

You can find all the Internet Protocols described in documents in Request For Comments (RFCs), the format in which they were originally issued, available online at: www.internic.net/std. The mail RFC is RFC821.txt.

You can find all the WWW protocols described in documents linked to from the URL www.w3.org/pub/WWW/Protocols/. A careful study of some of these documents will often answer any protocol questions you have.

You can find more information on the Java mail API at java.sun.com/products/javamail/.

If you write a simple Swing GUI around this mail-sending code, you’ve written a mailer program! It’s not that hard to get the RFCs for the POP3 and IMAP protocols and write the code to read and display incoming mail, too.

1. POP3 is “Post Office Protocol 3,” and IMAP is “Internet Mail Access Protocol,” different standards for the client end of mail systems. POP3 downloads mail and keeps it on the client. IMAP keeps the mail on the server, which is more convenient if you read it from several different computers.
A Server Socket in Java

This section shows a simple example of creating a server socket to listen for incoming requests. We could write the server side of a simple NTP server, but let’s try something a little more ambitious. It should be fairly clear at this point that HTTP is just another of the many protocols that use sockets to run over the Internet.

A web browser is a client program that sends requests through a socket to the HTTP port on a server and displays the data that the server sends back. A basic web browser can be written in a couple of hundred lines of code if you have a GUI component that renders HTML, which Java does.

A web server is a server program that waits for incoming requests on the HTTP port and acts on those to send the contents of local files back to the requestor. It can be implemented in just a few dozen lines of code.

Security of Network Programs—A Cautionary Tale!

Be very careful when you start developing networked programs on your computer. Before you try it at work, check if there is a company policy about network use. You can get fired for doing the wrong thing!

The problem is that any server sockets you create may be visible more widely than you intended. If you are running this at home, and you are not using a firewall, your server socket will be visible to the entire net. That’s like leaving the front door of your home wide open.

When I was developing the HTTP server in Java for this chapter, I left it running on my PC to test it. Someone’s automated port scanner script soon noticed my server, made an unauthorized connection to it, and issued this HTTP command:

```
GET /scripts/..%35c../winnt/system32/cmd.exe?/c+dir HTTP/1.0
```

This is an attempt to break out of the scripts directory, run a shell, and do a “dir” to see what’s on my system. Crackers will try to add their own backdoor on your computer where you’ll never find it. Then they can use your system whenever it’s on the net (they love cable modems) for such things as distributed denial of service attacks. My server was logging client requests, but not fulfilling them, so the nimrod was out of luck. But be careful out there; people are actively looking for systems to break into.

The example here is part of the code for a web server. This is the code that opens a server socket on the http port, port 80, and listens for requests from web browsers. We echo the requests, but don’t act on them.

The code is split into two classes to better show what’s happening. The first class is the main program. It instantiates a server socket on port 80 (use port 1080 if you’re on a Unix system without root access). The code then does an accept() on the server socket, waiting for client connections to come in. When one does come
in, the program creates a new object to deal with that one connection and invokes its getRequest() method.

```java
public class HTTPServer {
    public static void main(String a[]) throws Exception {
        final int httpd = 80;
        ServerSocket ssock = new ServerSocket(httpd);
        System.out.println("have opened port 80 locally");

        Socket sock = ssock.accept();
        System.out.println("client has made socket connection");

        OneConnection client = new OneConnection(sock);
        String s = client.getRequest();
    }
}
```

There are only two new lines of code in this server program. This line:

```java
ServerSocket ssock = new ServerSocket(httpd);
```

and this line:

```java
Socket sock = ssock.accept(); // on the server
```

The first line instantiates a server socket on the given port (httpd is an int with the value 80). The second line does an accept() on this server socket. It will block or wait here until some client somewhere on the net opens a connection to the same port, like this:

```java
clientSock = new Socket("somehost", 80); // on the client
```

At that point, the accept() method is able to complete, and it returns a new instance of a socket to the server. The rest of this conversation will be conducted over the new socket, thus freeing up the original socket to do another accept() and wait for another client. At the client end, the socket doesn’t appear to change.

In a real server, the code will loop around and accept another connection. We’ll get to that. Here is the second half of the code: the OneConnection class that the main program uses to do the work for a single client request.
import java.io.*;
import java.net.*;
class OneConnection {
    Socket sock;
    BufferedReader in = null;
    DataOutputStream out = null;

    OneConnection(Socket sock) throws Exception{
        this.sock = sock;
        in = new BufferedReader(
            new InputStreamReader( sock.getInputStream() ) );
        out = new DataOutputStream(sock.getOutputStream());
    }

    String getRequest() throws Exception {
        String s=null;
        while ( (s=in.readLine())!=null) {
            System.out.println("got: "+s);
        }
        return s;
    }
}

The constructor keeps a copy of the socket that leads back to the client and opens the input and output streams. Sockets always do I/O on bytes, not Unicode chars. HTTP is a line-oriented protocol. We push a BufferedReader onto the input stream so we can use the convenient readLine() method. DataInputStream has one of those too, but it is deprecated.

If you’re using a binary protocol, do everything with streams, not readers/writers. We wrap a DataOutputStream on the output side of the socket. We don’t write anything in this version of the program, but we will soon develop it and start writing.
Socket Protocols

The getRequest() method reads successive lines from the socket and echoes them on the server. How does it know when to stop reading lines? This is one of the tricky things with sockets—they cannot tell the difference between “end of input” and “there is more input, but it is delayed coming through the network.”

To cope with this inability to know when it’s done, socket protocols use one of three approaches:

• have the client precede each message by a number giving the length of the following message. Or use some other indication to end transmission, such as sending a blank line.

• have the client close its output stream, using sock.shutDownOutput(). That causes the next read at the server end to return -1.

• set a timeout on the socket, using sock.setSoTimeout(int ms). With this set to a non-zero amount, a read call on the input stream will block for only this amount of time. Then it will break out of it by throwing a java.net.SocketTimeoutException, but leaving the socket still valid for further use.

The third approach, using timeouts, is the least reliable because timeouts are always too long (wasting time) or too short (missing input). HTTP uses a mixture of approaches one and two.

Running the HTTP Server Program

Compile the code and then run the program. Make sure you run it on a computer that is not already running a web-server, otherwise it will find that it cannot claim port 80. If all is well, the program will print out:

java HTTPServer
have opened port 80 locally!

then it will block, waiting for an incoming request on the port. This is exactly what a webserver does: opens port 80 and waits for incoming socket connections from clients.
Here's the interesting part. You can make that connection using any web browser! Just start up your browser and direct it to the computer where you are running the Java program. You can run your browser on a different system altogether, and give it the name of the computer running the Java program. Or, if you are running everything on one computer, the name will be “localhost,” and the URL will be something like:

http://localhost/a/b/c/d.html

The rest of the URL doesn’t matter since our server program doesn’t (yet) do anything with the incoming request. You will see the Java server print out the message that a socket connection has been made (“got a socket”), and then print the HTTP text it receives on the socket from the browser!

got a socket
got: GET /a/b/c/d.html HTTP/1.1

got: Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
got: Accept-Language: en-us

got: Accept-Encoding: gzip, deflate

got: User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 4.0)
got: Host: localhost

got: Connection: Keep-Alive

got:
These strings are HTTP headers. They are created by the browser to tell the server what file it has asked for, and they provide information about what kinds of format the browser can accept back.

A couple more points to note here. First, almost all servers uses threads. That way, they can serve the client and at the same time accept further requests. We will shortly show the code to do this. Second, these dozen or so lines of server code are at the heart of every webserver. If you add a couple of routines to read whatever file the browser asks for and write it into the socket, you have written a webserver. Let’s do it.

The ServerSocket API is:

```java
public class ServerSocket {
    public ServerSocket() throws IOException;
    public ServerSocket(int) throws IOException;
    public ServerSocket(int, int) throws IOException;
    public ServerSocket(int, int, InetAddress) throws IOException;

    public Socket accept() throws java.io.IOException;
    public void close() throws java.io.IOException;
    public java.nio.channels.ServerSocketChannel getChannel();

    public void bind(SocketAddress) throws IOException;
    public void bind(SocketAddress, int) throws IOException;
    public boolean isBound();
    public InetAddress get InetAddress();
    public int getLocalPort();

    public boolean isClosed();
    public synchronized void setSoTimeout(int) throws SocketException;
    public synchronized int getSoTimeout() throws java.io.IOException;
    public static synchronized void setSocketFactory(SocketImplFactory)
        throws IOException;
    public synchronized void setReceiveBufferSize(int) throws
        SocketException;
    public synchronized int getReceiveBufferSize() throws SocketException;
}
```

The accept() method listens for a client trying to make a connection and accepts it. It creates a fresh socket for the server end of the connection, leaving the server socket free to do more accepts.

The bind() method is used to connect an existing socket to a particular IP address and port. You would use this when you want to use channels instead of streams for socket I/O; there’s an example at the end of the chapter.

The other methods should be clear from their names. There are other methods in the API, but these are the main ones you will use.
Debugging Sockets

The little HTTPServer program we just saw can be used to help debug some server socket problems. You can see exactly what headers the browser sends you for different HTML requests. It works for other protocols too. If you make the code listen on another port, you can look at the incoming stream there.

Standing on the Corner, Watching All the Packets Go By

The server program, shown on the previous pages, will echo all the input that is sent to one socket. This is similar to the way that the FBI’s controversial Carnivore program works.

Carnivore was created so that the FBI could do the online equivalent of phone tapping. It works at the more fundamental level of individual packets rather than sockets, but the principle is the same.

Carnivore is basically a packet sniffer that can be installed at an ISP and directed to copy packets that meet certain criteria (to or from a given IP address, for example). In this way, Carnivore can give the FBI a copy of all the email, all the web site visits, all the telnet sessions for a particular target over the course of a month or more. A court order is needed to authorize each use of Carnivore.

The FBI made a PR error by giving the program such an aggressive name. Law enforcement needs access to these tools to track down online fraud, network disruption, and other crimes. But they would have done themselves a favor by calling the software something calmer like “Old Packet Collector.”

Another debugging technique uses the telnet program to look at incoming text to a client socket. Telnet’s actual purpose is to open a command shell on a remote computer. The lines you type are sent over the socket connection, and the responses sent back the same way. However, you can tell telnet to use any port. The stream that it receives on that port will be displayed in the telnet window, and the things you type will be sent through the socket back to the server. The characters you type will be sent to the other end, but not echoed however.

Telnet is just a quick and dirty debugging technique to help you see what’s going on. Figure 17-4 uses telnet to see what an NTP server is sending back. Most servers will close a socket as soon as they have given you the requested information, hence the “connection lost” pop-up window. There is also a “keep-alive” option to a socket that requests the connection be retained for expected use in the very near future. This is useful for HTTP.
These days you should avoid the use of telnet and ftp for their main purpose, as they send passwords “in the clear” to the remote socket. They are thus vulnerable to packet-sniffing by crackers at routers. Use SSH, the secure shell, instead. There is a Java implementation of SSH on the CD that comes with this book.

Finally, there’s a very helpful website at straylight.cso.niu.edu. You can use one of their webpages (specifically, straylight.cso.niu.edu/cgi-bin/test-cgi.cmd) to see what is happening with your HTML pages. If you specify that web page as the “Action” value for an HTML form, when you press the “submit” button, the script will echo back to you everything that your form sent across. If this site goes
Getting the HTTP Command

Let's add a few lines of code (in bold) to our server to extract the HTTP “GET” command that says what file the browser is looking for. We will develop this example by extending the OneConnection class. That way, we will add just the new code in the child class, and use the existing methods from the parent. The code in the new child class is:

```java
class OneConnection_A extends OneConnection {

    OneConnection_A(Socket sock) throws Exception {
        super(sock);
    }

    String getRequest() throws Exception {
        String s=null;
        while ( (s=in.readLine())!=null) {
            System.out.println("got: "+s);
            if (s.indexOf("GET") > -1) {
                out.writeBytes("HTTP-1.0 200 OK\r\n");
                s = s.substring(4);
                int i = s.indexOf(" ");
                System.out.println("file: "+ s.substring(0, i));
                return s.substring(0, i);
            }
        }
        return null;
    }
}
```

The `getRequest()` method now looks at incoming HTTP headers to find the one containing a GET command. When it finds it, it writes an acknowledgement back to the browser (the “200 OK” line), and extracts the filename from the GET header. The filename is the return value of the method.

The main program will need to construct the `OneConnection_A` object and then call its `getRequest()` method. From here it is a small step to actually get that file and write it into the socket.
Here’s a new class that is a child of OneConnection_A; it adds a method to get the file of the given name and write it into the socket. Since it knows how big the file is, it might as well generate the HTTP header that gives that information.

```java
class OneConnection_B extends OneConnection_A {

    OneConnection_B(Socket sock) throws Exception {
        super(sock);
    }

    void sendFile(String fname) throws Exception {
        String where = "\tmp\" + fname;
        if (where.indexOf("\..") > -1)
            throw new SecurityException("No access to parent dirs");
        System.out.println("looking for " + where);
        File f = new File(where);
        DataInputStream din = new DataInputStream(
                        new FileInputStream(f) );
        int len = (int) f.length();
        byte[] buf = new byte[len];
        din.readFully(buf);
        out.writeBytes("Content-Length: " + len + "\r\n\n");
        out.writeBytes("Content-Type: text/html\r\n\n");
        out.write(buf);
        out.flush();
        out.close();
    }
}
```

Never Use println() with Sockets!

The println() method is defined to output the platform specific line separator. This will be "\n" on Unix, "\r" on Macs, and "\r\n" on Windows. However, lots of TCP/IP protocols are line based, and the line is defined to end with carriage return line feed "\r\n", or just line feed "\n". So if you’re on a Mac, the println method won’t output something that a socket server recognizes as a complete end of line sequence.

The Mac client will do a println, which sends a "\r", and then waits for a response from the server. The server will get the "\r" and wait for a "\n" to complete the end of line sequence. Result: deadlock! Each end is waiting for something from the other. See Apple Tech Note 1157 for more on this: developer.apple.com/technotes/tn/tn1157.html

The solution is to never use println with remote protocols on any platform. Always use explicit "\r\n" characters when writing to a socket.
The main program will need to construct the OneConnection_B object and then add a call to its send file method. Now that our server has the ability to return files we need to build in some security. The first few lines of the method prepend the string “/tmp/” onto the filename. The code also checks that the filename does not contain the string “..” to enter a parent directory. These two limitations together ensure that the server will only return files from your \tmp directory.

The “Content-Length” and “Content-Type” are two standard HTTP headers that help the browser deal with what you send it. The blank line tells the browser that is the end of the headers and the text that follows should be displayed.

At this point you should try compiling the code, placing a test html file in the \tmp directory, and then starting the server. Browse the URL localhost/tmp/example.html and check that the browser displays the file correctly.

We have completed a basic webserver. That’s quite an accomplishment! The next section looks at client side sockets again, in particular how to use a socket to pull information from a web page. It then shows the same task done by a URLConnection. We then describe the class that represents IP addresses and finish the chapter by making the web server multithreaded.
HTTP and Web Browsing: Retrieving HTTP Pages

Here is an example of interacting with an HTTP server to retrieve a web page from a system on the network. This shows how easy it is to post information to HTML forms. Forms are covered in more depth in the chapter on servlets, and you may want to refresh your memory on that section.

For the impatient, HTML forms allow you to type some information in your browser which is sent back to the server for processing. The information may be encoded as part of the URL, or sent separately in name/value pairs.

The Yahoo site is a wide-ranging access portal. They offer online stock quotes that you can read in your browser. I happen to know (by looking at the URL field of my browser) that a request for a stock quote for ABCD is translated to a socket connection of:

http://finance.yahoo.com/q?s=abcd

That’s equivalent to opening a socket on port 80 of finance.yahoo.com and sending a “get /q?s=abcd.” You can make that self same request yourself, in either of two ways. You can open a socket connection to port 80, the http port. Or you can open a URL connection which offers a simpler, higher-level interface. We’ll show both of these here. Here’s the stock finder done with sockets:
import java.io.*;
import java.net.*;
public class Stock {

    public static void main(String a[]) throws Exception {
        if (a.length!=1) {
            System.out.println("usage: java Stock <symbol> ");
            System.exit(0);
        }

        String yahoo = "finance.yahoo.com";
        final int httpd = 80;
        Socket sock = new Socket(yahoo, httpd);

        BufferedWriter out =
            new BufferedWriter( new OutputStreamWriter( sock.getOutputStream() ) );

        String cmd = "GET /q?" +"s=" +a[0] +"\n";
        out.write(cmd);
        out.flush();

        BufferedReader in =new BufferedReader(
            new InputStreamReader( sock.getInputStream() ) );

        String s=null;
        int i, j;
        // pick out the stock price from the pile of HTML
        // it's in bold, so get the number following "<b>"
        while ( (s=in.readLine()) != null)  {
            if (s.length()<25) continue;
            if ((i=s.indexOf(a[0].toUpperCase())) < 0) continue;
            s=s.substring(i);
            if ((i=s.indexOf("<b>")) < 0) continue;
            j = s.indexOf("</b>");
            s=s.substring(i+3,j);
            System.out.println(a[0] +" is at "+s);
            break;
        }
    }
}

The Yahoo page that returns stock quotes contains thousands of characters of
hrefs to ads and formatting information. It consists of many HTML lines like this:

    a href="/q?s=SUNW&d=t">SUNW</a></td><td nowrap
    align=center>12:03PM</td><td nowrap><b>9.55</b>
Luckily it’s fairly easy to pull out the stock price. From inspecting the output, it’s on a line with more than 25 chars. The line contains the stock symbol rewritten in upper case. The number we want is bracketed by <b> ... </b>, which is the HTML to print the number in bold face.

Given all that, running the program provides this output:

date Stock ibm
ibm is at 101.26

It was a lot more fun running this program in the year 2000. Here is the same program, rewritten to use the classes URL and URLConnection. Obviously, URL represents a URL, and URLConnection represents a socket connection to that URL. The code to do the same work as before, but using URLConnection is:

date import java.io.*;
date import java.net.*;
date public class Stock2 {
date   public static void main(String a[]) throws Exception {
date     if (a.length!=1) {
date         System.out.println("usage: java Stock <symbol> ");
date         System.exit(0);
date     }

date     String yahoo = "http://finance.yahoo.com/q";

date     URL url = new URL(yahoo);
date     URLConnection conn = url.openConnection();
date     conn.setDoOutput(true);
date     PrintWriter pw = new PrintWriter( conn.getOutputStream());
date     pw.print("s=" + a[0]);
date     pw.close();

date     BufferedReader in = new BufferedReader( new InputStreamReader( conn.getInputStream()));

date     String s=null, stock=a[0].toUpperCase();
date     int i=0,j=0;
date     while ( (s=in.readLine()) != null) {
date         if (s.length()<25) continue;
date         if ((i=s.indexOf(a[0].toUpperCase())) < 0) continue;
date         s=s.substring(i);
date         if ((i=s.indexOf("<b>")) < 0) continue;
date         j = s.indexOf("</b>");
date         s=s.substring(i+3,j);
date         System.out.println(a[0] +" is at "+s);
date         break;

date     }

date   }

date}
The main difference here is that we form a URL for the site and file (script) that we want to reference. Then we open a connection to the URL, tell it that we are going to do output to it, and write the “name=value” parameter. We finish up as before, reading what the socket writes back and extracting the characters of interest. Clearly, this program will stop working when Yahoo changes the format of the page, but it demonstrates how we can use a URL and URLConnection for a slightly higher-level interface than a socket connection. We could even go one step further and use the class HttpURLConnection which is a subclass of URLConnection. Please look at the HTML documentation for information on these classes.

A URL can pose a security risk, since you can pass along information even by reading a URL. Requesting www.cia.gov/cgi-bin/cgi.exe/secretinfo passes “secret-info” along to the CGI script, for example. Since requesting a URL can send out information just as a Socket can, requesting a URL has the same security model as access to Sockets. Namely, an applet can only open a socket connection back to the server from which the applet came. Security is also the reason that an applet may not open a socket connection to any other system except its server. Otherwise, it could look at information on its subnet behind the firewall, and send it back to crackers everywhere.

If you are behind a firewall (and who isn’t these days?) you will need to tell Java the details of your proxy server and port in order to access hosts outside the firewall. You do this by defining properties, perhaps when starting the code:

```java
java -DproxySet=true -DproxyHost=SOMEHOST -DproxyPort=SOMENUM code.java
```

Without this, you’ll get an UnknownHostException. The proxy settings are needed for both java.net.URLConnection and for java.net.Sockets. At work, your systems administrator will know the values. At home, you won’t be using a proxy server unless you set it up yourself.

**How to Find the IP Address Given to a Machine Name**

The class java.net.InetAddress represents IP addresses and about one dozen common operations on them. The class should have been called IP or IPAddress, but was not (presumably because such a name does not match the coding conventions for classnames). Common operations on IP addresses are things like: turning an IP address into the characters that represent the corresponding domain name, turning a host name into an IP address, determining if a given address belongs to the system you are currently executing on, and so on.
Just Java 2

InetAddress has two subclasses:

- **Inet4Address**
  The class that represents classic, version 4, 32-bit IP addresses

- **Inet6Address**
  The class that represents version 6 128-bit IP addresses

Your programs will not use these classes directly very much, as you can create sockets using domain and host names. Further, in most of the places where a host-name is expected (such as in a URL), a String that contains an IP address will work equally well. However, if native code passes you an IP address, these classes give you a way to work on it.

The InetAddress class does not have any public constructors. Applications should use the methods getLocalHost(), getByName(), or getAllByName() to create a new InetAddress instance. The program that follows show examples of each of these.

This code will be able to find the IP address of all computers it knows about. That may mean all systems that have an entry in the local hosts table, or (if it is served by a name server) the domain of the name server, which could be as extensive as a large subnet or the entire organization.

```java
import java.io.*;
import java.net.*;
public class addr {
    public static void main(String a[]) throws Exception {
        InetAddress me = InetAddress.getByName("localhost");
        PrintStream o = System.out;
        o.println("localhost by name =" + me);

        InetAddress me2 = InetAddress.getLocalHost();
        o.println("localhost by getLocalHost =" + me2);

        InetAddress[] many = InetAddress.getAllByName("microsoft.com");
        for (int i=0; i<many.length; i++)
            o.println( many[i] );
    }
}
```
Run it with:

```java
java addr
```

localhost by name =localhost/127.0.0.1
localhost by getLocalHost =zap/10.0.10.175
Microsoft: microsoft.com/207.46.230.218
Microsoft: microsoft.com/207.46.230.219
Microsoft: microsoft.com/207.46.197.100
Microsoft: microsoft.com/207.46.197.101
Microsoft: microsoft.com/207.46.197.102

The `getAllByName()` method reports all the IP addresses associated with a domain name. You can see from the output above that Microsoft.com, like most big sites, is served by multiple IP addresses, on two different subnets (probably for fault tolerance). Each of those five IP addresses probably represents load balancer hardware fanning out to dozens of server nodes.

**Some Notes on Protocol and Content Handlers**

Some of the Java documentation makes a big production about support for extending the MIME types known to browsers. If it is asked to browse some data whose type it doesn’t recognize, it can simply download the code for the appropriate handler based on the name of the datatype and use that to grok the data. This is exactly what happens with plug-ins. If you stumble across a RealAudio file, the browser prompts you to download the plug-in that can play it. Java can make this completely automatic. Or so the theory runs. It hasn’t yet been used much in practice.

The theory of the handlers is this. There are two kinds of handler that you can write: protocol handlers and content handlers.

A **protocol handler** talks to other programs. Both ends follow the same protocol in order to communicate structured data between themselves (“After you,” “No, I insist—after you.”) If you wrote an Oracle database protocol handler, it would deal with SQL queries to pull data out of an Oracle database.

A **content handler** knows how to decode the contents of a file. It handles data (think of it as the contents of something pointed to by a URL). It gets the bytes and assembles them into an object. If you wrote an MPEG content handler, it would then be able to play MPEG movies in your browser, once you had brought the data over there. Bringing MPEG data to your browser could be done using the FTP protocol, or you might wish to write your own high performance protocol handler.

Content handlers and protocol handlers may be particularly convenient for web browsers, and they may also be useful in stand-alone applications. There is not a lot of practical experience with these handlers in Java yet, so it is hard to offer definitive advice about their use. Some people predict they are going to be very important for decoding file formats of arbitrary kinds, while other people are ready to be convinced by an existence proof.
How to Make an Applet Write a File on the Server

Everyone who writes an applet sooner or later wants to have it write a file back on the server. This is perhaps the most frequently asked question in Java. You might want the high score in a high score file, or have a guestbook that visitors to the page can sign, or count the number of HTTP requests for this page.

If you think about it, giving write access to your server system to anyone anywhere on the Internet is a very bad idea. It would be the equivalent of leaving the keys in your car with the engine running and a sign on the windshield saying, “Help yourself.” One of Java’s goals is to improve, not undermine, system security and so there is no built-in support allowing an applet to directly write to a file on a server. None! You can read a file on the server inside an applet, but you absolutely cannot directly create or write a file on the server unless the server does it for you. But all is not lost. We could write the ping program even though Java doesn’t support ICMP.

How can you persuade the server to write a file for you? One possibility is to write a system demon that runs on the server 24 hours a day and listens to a specified port. When one of your applets wants to write a file, it opens a socket on that port on the server, sends commands saying what it wants to do, gives the filename, and then sends the data. This is open to abuse, though. Anyone who knows about this service could similarly connect to that port and fill up your file system with their files. You could lessen the probability of that happening if you protected the service by a password challenge.
The Linlyn Class for File Transfer

Well, what do you know? The previous paragraph is a pretty good description of the standard FTP (file transfer protocol) service. This service will frequently be running on any system that has full TCP/IP support. The standard FTP port is port 21 for commands and port 20 for data. Simply open a socket connection from your applet to port 21 on the server and you can have FTP write the files for you. Of course, you have to know the FTP commands to send, and you have to be able to deal with the password challenge.

Luckily, that work has already been done for you. The request for code to write to the server from an applet is made so frequently that I got together with my colleague, Bob Lynch, and we created the Linlyn class. (Hey, at least the name’s not an acronym.) The Linlyn class does easy FTP file transfer from an applet to a server. The Linlyn class was designed with absolute ease-of-use as its foremost consideration, and a copy of it (along with a sample applet) is on the CD in the goodies directory.

The Linlyn class is published under the GNU public license for anyone to use or improve. Be advised that in its standard form you compile the userid and password as a String into the class file. That is appropriate only for use on a secure intranet, and must not be used for an applet that you publish on the Internet. It would be a simple matter, however, for you to prompt the user for the password, instead of hard-coding it in the program. Then you or anyone that you trust with your FTP password (and your credit card, car keys, wallet, etc.) can write files on your server from your applet.
A Multithreaded HTTP Server

There’s one improvement that is customary in servers, and we will make it here. For all but the smallest of servers, it is usual to spawn a new thread to handle each request. This has three big advantages:

1. Foremost, it makes the server scalable. The server can accept new requests independent of its speed in handling them. (Of course, you better buy a server that has the mippage to keep up with requests.)

2. By handling each request in a new thread, clients do not have to wait for every request ahead of them to be served.

3. The program source can be better organized, as the server processing is written in a different class.

The following code demonstrates how we would modify our HTTP web server to a new thread for each client request. The first step is to make another child in the One_Connection hierarchy to implement the Runnable interface. Give it a run method that will actually do all the work: get the request, then send the file.

```java
import java.io.*;
import java.net.*;
class OneConnection_C extends OneConnection_B
    implements Runnable {
    OneConnection_C(Socket sock) throws Exception {
        super(sock);
    }

    public void run() {
        try {
            String filename = getRequest();
            sendFile(filename);
        } catch (Exception e) {
            System.out.println("Excpn: ", e);
        }
    }
}
```
The main program will have the server socket and it will put the accept in a loop (so that we can handle many requests, not just the first one). It will instantiate our connection class as before, turn it into a thread, and invoke its start method. The code follows:

```java
public class HTTPServer4 {
    public static void main(String a[]) throws Exception {
        final int httpd = 80;
        ServerSocket ssock = new ServerSocket(httpd);
        while (true) {
            Socket sock = ssock.accept();
            System.out.println("client has made socket connection");
            OneConnection_C client = new OneConnection_C(sock);
            new Thread(client).start();
        }
    }
}
```

The code seems so brief because we have draped the functionality across several classes in an inheritance hierarchy. That was done so that the example code would be smaller and easier to present. You should try putting the code back into one or two classes. It’s still only 50 or 60 lines long. This has got to be the world record for the smallest HTTP server.
A Mapped I/O HTTP Server

The final section of this chapter presents the code to use the new mapped I/O facility in a socket server. Sun distributes a sample mapped I/O socket application with JDK 1.4 beta. However, their sample application does not work on Windows 98. It hangs in the accept() statement.

I reported the bug to Sun and hope it will be fixed in the final release. It may not be: after all, mapped I/O is a server feature, and Windows 9x is not a server operating system. Anyway, as a refresher, an example of channel I/O on files would be this program that duplicates a file:

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
class Files {

    void copyThruChannel(String fname) throws Exception {
        File f = new File(fname);
        FileInputStream fin = new FileInputStream(f);
        int len = (int) f.length();

        FileChannel fc = fin.getChannel();
        System.out.println("allocating buff");
        ByteBuffer myBB = ByteBuffer.allocate(len);
        int bytesRead = fc.read(myBB);
        myBB.flip();

        System.out.println("getting fout channel");
        FileOutputStream fos = new FileOutputStream(fname + ".copy.txt");
        FileChannel fco = fos.getChannel();
        int bytesWritten = fco.write(myBB);
        fco.close();
    }

    public static void main(String a[]) throws Exception {
        Files client = new Files();
        client.copyThruChannel(a[0]);
    }
}
```

In a similar way, the code to update our HTTP server, so that it uses channel I/O, looks like this:
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.net.*;
class OneConnection_D extends OneConnection_C {
    OneConnection_D(Socket sock) throws Exception {
        super(sock);
    }
    void sendThruChannel(String fname) throws Exception {
        File f = new File(fname);
        FileInputStream fin = new FileInputStream(f);
        int len = (int) f.length();
        FileChannel fc = fin.getChannel();
        System.out.println("allocating buff");
        ByteBuffer myBB = ByteBuffer.allocate(len);
        int bytesRead = fc.read(myBB);
        myBB.flip();
        System.out.println("getting sock channel");
        SocketChannel sc = sock.getChannel();
        int bytesWritten = sc.write(myBB);
        sc.close();
    }
}
public class HTTPServer4 {
    public static void main(String a[]) throws Exception {
        final int httpd = 80;
        ServerSocketChannel ssc = ServerSocketChannel.open();
        InetSocketAddress isa = new InetSocketAddress(InetAddress.getLocalHost(), httpd);
        ssc.socket().bind(isa);
        System.out.println("have opened port 80 locally!");
        System.out.println("waiting for accept");
        Socket sock = ssc.accept();
        System.out.println("client has made socket connection");
        OneConnection_D client = new OneConnection_D(sock);
        String filename = client.getRequest();
        client.sendThruChannel(filename);
    }
}

Note: This version is not multithreaded to keep the code focused on the issue of interest. The main routine shows how you have to open a server socket channel, then bind it to the port of interest. From here it is easy to use mapped I/O.
Further Reading


The modest title hides the fact that this book will be useful to a wider audience than just network administrators. It is a very good practical guide to TCP/IP written as a tutorial introduction.


When a book starts off with an apology for the dullness of the subject material, you just know that the author has some unusual ways about him.


The canonical guide to network programming.
Exercises

1. Extend the previous example mail program so that it prompts for user input and generally provides a friendly front end to sending mail.

2. Write a socket server program that simply returns the time on the current system. Write a client that calls the server and sends you mail to report on how far apart the time on the local system is versus the time on the current system.

3. In the previous exercise, the server can only state what time it is at the instant the request reaches it, but that answer will take a certain amount of time to travel back to the client. Devise a strategy to minimize or correct for errors due to transmission time. (Hard—use a heuristic to make a good guess.)

4. Read the API for java.net.URLEncoder and URLDecoder and write a program that encodes a string into the MIME format called \texttt{x-www-form-urlencoded}.

5. Update the multithreaded webserver so that it can also serve JPG and GIF files and correctly identify their type to the browser. You can just use the file extension as an indicator of the contents.
Some Light Relief—Using Java to Stuff an Online Poll

The email to me was brief. It just read:

From billg@Central Mon May  4 11:57:41 PDT
Subject: Hank the Angry Dwarf
To: jokes@Sun.COM

Hey everyone. If you’ve got five seconds to spare, go to the following url:


and vote for:

Hank the Angry, Drunken Dwarf

This is a huge joke. We want to try to get Hank way up there on the People Magazine 50 most beautiful people of the year list. As of 2:00 AM, he’s already up to number 5!

Well, I can recognize a high priority when I see one. I put down the critical bug fix I was working on, went right to the website, and checked what this was all about.

Every year the celebrity gossip magazine People prints a list of “the 50 most beautiful people in the world,” and this year they were soliciting votes on their website. People had started the ball rolling with nominations for actors like Kate Winslet and Leonardo DiCaprio, who were in the public eye because of their roles in the Titanic movie.

People magazine gave web surfers the opportunity to write in names of people for whom they wanted to vote. A fan of the Howard Stern radio show nominated “Hank the angry, drunken dwarf” for People’s list. When Stern heard about Hank’s nomination as one of the most beautiful people in the world, he started plugging the candidacy on the radio. A similar phenomenon took place on the Internet, and many people received email like I did. Another write-in stealth candidate widely favored by netizens was flamboyant, blond-haired, veteran pro-wrestler Ric Flair.
Hank, the angry, drunken dwarf, is an occasional guest on Stern’s syndicated radio program. Hank is a 36-year old dwarf who lives in Boston with his mother and has made a name for himself as a belligerent, if diminutive, devotee of beer, tequila, and Pamela Anderson.

The People website soon crashed under the strain of incoming votes for Hank. When the People poll closed, the results were as follows:

- 230,169 votes Hank the dwarf: Angry, drunken dwarf and Stern radio guest
- 17,145 votes Ric Flair: 25-year pro-wrestling performer
- 14,471 votes Leonardo DiCaprio: High school dropout
- 7,057 votes Gillian Anderson: Actress
- 5,941 votes Kate Winslet: High school dropout

Hank Nassif, the angry, drunken dwarf, was officially the most beautiful person in the world, by a margin of more than 10-to-1 over the runner-up! Unhappily, People magazine showed their true colors at this point, ignored the clear mandate from the website, and went ahead with a cover story naming the guy who came in third as the official “most beautiful person in the world” for 1998. What a rip-off.

**The Java Votebot**

There were dark allegations here of automated voting programs, or votebots. I was shocked. But not too shocked to show you how to write a votebot using your new-found Java networking skills.

First, find any online poll. Lots of sites run them because they are a lot cheaper than having actual meaningful content. Let’s pick on, I don’t know, say, CNN.com. The goofballs in the media are always running some kind of “scientific” poll in an attempt to seem “with it” and “hip” to the latest trends and “cool slang.” In October 2001, they were running a poll asking, “Is al Qaeda sending coded messages to followers via video statements?” You could answer “yes” or “no.” There wasn’t a box for people to respond, “This kind of inane question only trivializes serious matters, and distracts attention from the real issues”.

Do a “view source” on a poll web page in your browser to see how they are submitting the results. You’re probably going to want to reread the chapter on servlets to get the most out of this. The part of the HTML page that deals with the poll will probably look something like this:
Just Java 2

So that tells us this is a simple form which is posted to URL “http://poll.cnn.com/,” the script is called “poll,” its argument is called “1682781,” and the name/value posted is “Answer168279=1” for yes, and “Answer168279=2” for no. There is a hidden attribute giving the question number, too.

It doesn’t matter which way you stuff this poll, the point is that a news organization needs to decide if it is in the news business or the entertainment business. Here’s the program to do it:
// implements a votebot to stuff Internet polls
import java.io.*;
import java.net.*;
public class votebot {
    public static void main (String args[]) {
        try {
            for (int i=0; i<1000; i++) {
                URL u = new URL("http://poll.cnn.com");
                URLConnection uc = u.openConnection();
                uc.setDoOutput(true);
                OutputStream os = uc.getOutputStream();
                PrintStream ps = new PrintStream(os);
                ps.print("GET /poll?1682781p\r\n\n");
                ps.print("Question=1&Answer168279=2\r\n\n");
                System.out.print(".");
            }
        } catch (Exception ex) {
            System.out.println("Excpn: "+ex.getMessage());
        }
    }
}

If this doesn’t work for you, there are several possible reasons. There could be
a bug in the code. Or the polling site may be employing “electronic countermea-
sures,” such as discarding multiple inputs from one IP address. Or the “poll” may
in fact be completely fake and discard all input from anyone at all times.

After writing several more test programs, I reached the conclusion that the
“results” of the CNN online polling are completely unrelated to the web votes
cast!