Chapter 3

Java Sockets and URLs

- Sockets and Interprocess Communication
- Client/Server Methodology
- The Pizza Order Protocol (TPOP)
- The TPOP Server
- The TPOP Client
- UDP Client

Sockets and Interprocess Communication

At the heart of everything we discuss in this book is the notion of interprocess communication (IPC). In this chapter, we will look at some examples using Java mechanisms for interprocess communication. IPC is a fancy way of saying "two or more Java programs talking with each other." Usually the programs execute on different computers, but sometimes they may execute on the same host.

Introduction to IPC

When you call Charles Schwab to check on your stock portfolio, you dial a telephone number. Once connected, you press some telephone buttons to request various services and press other buttons to send parameters, such as the numeric codes for stock symbols in which you are interested. You may think of your account as an object with different methods that you can invoke to purchase or to sell stocks, to get current quotes, to get your current position in a stock, or to request a wire transfer to a Swiss bank. You are a *client* and the other end is a *server*, providing the services (methods) you request.

Of course, the server also provides services to many other clients. You can be a client of other servers, such as when you order a pizza with a push button telephone. Sometimes a server can be a client as well. A medical records query server may have to send a request to two or three hospitals to gather the information you

request for a patient. Thus your server becomes a client of the hospital servers it queries on your behalf.

All these situations are examples of interprocess communication. Each client and each server reside in different processes. Sometimes you, the individual, are the client; other times it is a computer. Sometimes the server is an application that listens in on what you type on your telephone pad and processes the information; other times it will be a program, perhaps written in Java as we will do later in this chapter. IPC is how our applications communicate, but it also refers to the mechanism we use. This chapter explores the fundamentals of IPC using something called a socket.

Sockets. The communication construct underneath all this communication is more than likely a **socket**. Each program reads from and writes to a socket in much the same way that you open, read, write to, and close a file. Essentially, there are two types of sockets:

- One is analogous to a telephone (a connection-oriented service, e.g., Transmission Control Protocol)
- One is analogous to a mailbox (a connectionless "datagram" service, e.g., User Datagram Protocol)

An important difference between TCP connection sockets and UDP datagram sockets is that TCP makes sure that everything you send gets to the intended destination; UDP, on the other hand, does not. Much like mailing a letter, it is up to you, the sender, to check that the recipient received it. The difference between the two protocols is very similar to comparing the differences between using the phone to talk to friends and writing them letters.

When we call a friend using a telephone, we know at all times the status of the communication. If the phone rings busy, we know that we have to try later; if someone answers the phone, we have made a connection and are initiating the message transfer; if the person that answered the phone is the right person, we talk to them thereby transferring whatever information we intended to deliver.

Had we written a letter, we know that we would have initiated an information transfer after we dropped it off at the mailbox. This is where our knowledge of the transfer, in most cases, ends. If we get a letter back and it starts out with "Thanks for your letter," we know that our letter was received. If we never again hear from the person, there is some doubt that they ever received our letter.

Sometimes when you use the postal service, your letter becomes "lost in the mail." When the letter absolutely, positively has to be there, you may need a more reliable form of postage. Similarly, your choice between using a datagram or a connection

socket is easily determined by the nature of your application. If all your data fits in an 8K datagram and you do not need to know if it was received at the other end, then a UDP datagram is fine. Mailing party invitations is one example where UDP is more appropriate than TCP. If the length of service warrants the expense of establishing a connection (three handshake packets), or it is necessary that all the packets be received in the same order as they were sent, such as transferring a file that is more than 8K bytes long, then a TCP socket must be used. Likewise, if we were to mail our important package using something like Federal Express, we would be able to track the package and know when it arrives at its destination.

Here is another way to look at this. Suppose we have a server that is somewhere on the network but we don't know where. To communicate with this type of server, we must first announce our presence, listen for an answer, and then carry on the conversation in lockstep where first one end sends then listens while the other end listens then talks. This is like a student walking into the reserve room of a college library and, upon not seeing the librarian right away, saying, "Is there anyone here?" and then listening for a response.

"Good afternoon, I'll be with you in a moment."

"I'd like the book Prof. Steflik put on reserve for CS-341."

"Here it is. Please leave your Student ID card."

We announced our presence and started listening. The server was listening, heard us, replied with an implied go ahead, and returned to listening. We heard the server's response, announced what we wanted, and returned to listening. The server (librarian) heard our request, retrieved the information (the book), and delivered it. This back and forth type of communication is known as half duplex, where only one endpoint talks at a time; contrast this with full duplex, where both endpoints can talk and listen at the same time.

NOTE: A socket is sometimes called a "pipe" because there are two ends (or points as we occasionally refer to them) to the communication. Messages can be sent from either end. The difference, as we will soon see, between a client and a server socket is that client sockets must know beforehand that information is coming, whereas server sockets can simply wait for information to come to them. It's sort of like the difference between being recruited for a job and actively seeking one.

In this chapter, we will write an online ordering application, using TCP, and a broadcast communication application, using UDP. These applications will use the following classes from the *java.net* package, as illustrated in Table 3-1.

Table 3–1 **Java.net.* Types and Their Corresponding Protocol**

Mechanism	Description
Socket	TCP endpoint (a "telephone")
ServerSocket	TCP endpoint (a "receptionist")
DatagramSocket	UDP endpoint (a "mailbox")
DatagramPacket	UDP packet (a "letter")
URL	Uniform Resource Locator (an "address")
URLConnection	An active connection to an Internet object (e.g., a CGI-bin script, a DayTime service)

What Are Sockets? At the root of all TCP and UDP communications is a virtual device called a socket or a port; the terms are pretty much interchangeable. Sockets are a visualization mechanism for a software buffering scheme that is implemented deep in the bowels of the transport layer of the TCP/IP stack. The term "socket" actually comes from the old-fashioned telephone switchboard that Lily Tomlin's character Ernestine, the telephone operator, uses. The concept is pretty similar: Each socket in the switchboard represents a person or service that an incoming call can be routed to; when an incoming call is answered, the operator connects it to the appropriate socket, thereby completing the connection between the client (the caller) and the server (person being called). In the telephone switchboard each socket represented a specific person or service; in TCP/IP certain sockets are dedicated to specific agreed-upon services.

If we were to look at the packet level, we would see that a socket is really identified by a 16-bit number thereby giving us about 65,000 possible sockets. The first 1024 sockets are dedicated to specific agreed-upon services and are therefore called well-known ports. For each of the services provided on the well-known ports, there is a corresponding protocol that defines the manner in which clients and servers using that port should communicate. The protocols themselves are arrived at through a process known as the RFC process. Table 3-2 lists some of the more common TCP/IP services, their "well-known" ports, and their respective RFCs. Every Internet standard starts out as a "Request for Comment" or RFC. Through an interactive process an RFC, if "worthy," will be refined and developed by the Internet community into a standard.

Exploring Some of the Standard Protocols. When starting to understand sockets programming, it's always best to start out by examining the "trivial" protocols first and then move on to the more complex and finally to our own, application-specific protocols. The trivial protocols are a subset of Internet protocols that are simple, straightforward, and easy to implement.

Table 3–2 **Some Well-Known Port Services**

Port	Protocol	RFC
13	DayTime	RFC 867
7	Echo	RFC 862
25	SMTP (e-mail)	RFC 821 (SMTP) RFC 1869 (Extnd SMTP) RFC 822 (Mail Format) RFC 1521 (MIME)
110	Post Office Protocol	RFC 1725
20	File Transfer Protocol (data)	RFC 959
80	Hypertext Transfer Protocol	RFC 2616

Daytime. The Daytime service is usually provided on TCP and UDP port 13. Assuming that we have the address of a host that is running the Daytime service, the operation is straightforward. Using TCP the client connects to the Daytime port (13) on the remote host; the remote host accepts the connection, returns its current date and time, and closes the connection. This can be easily demonstrated using the Windows 95 Telnet client. Open up the Telnet client and click on Connect and the Remote System. Enter in the address of your host that provides the Daytime service, select the Daytime port, and click Connect. Notice that a date/timestamp is displayed in the client area and that a small dialog box indicates that the connection to the host has been lost.

This example is trivial but illustrates two things: First, the Windows Telnet client can be used to explore standard TCP-based protocols (we'll see this later with other protocols. Second, we really did demonstrate how the client end of the protocol works; the client makes a connection to the server, the server sends the timestamp and closes the connection, and, finally, the client receives the timestamp. To implement our own client, understanding what the client needs to do makes the task quite simple. A high-level design is

Create a socket

Create an input stream and tie it to the socket

Read the data from the input stream and display the result

To create a socket, define a variable for the socket class and initialize it using the class constructor:

```
Socket s = Socket("localhost", 7);
```

"localhost" is the name assigned to address 127.0.0.1 in your hostsfile; address 127.0.0.1 is known traditionally as your machine's "loop back port," and lets your

machine talk to itself. The line above creates a socket named "s" and connects it to port 7 on your loop back port. To connect to the Daytime service on any other host, just replace localhost with a string containing the dotted decimal name or IP address of whatever host you want to connect to.

This single instruction will create the socket object and attempt to connect it to the specified host. Because this has a possibility of failing (throwing an exception—a connection may not be established), we need to code it in a try/catch construct.

```
import java.io.*;
import java.net.*;
public class DayTimeClient{
   public static final port = 13;
   public static void main(String args[])
   {
      Socket s = null;
      String timestamp;
      try
      {
         // create the socket to the remote host
         s = new Socket(args[0], port);
         // create an input stream and tie it to the socket
         InputStream in = s.getInputStream();
         BufferedReader in =
            new BufferedReader(new InputStreamReader(in));
         // tell user they are connected
         System.out.println("Connected to : " +
            s.getInetAddress() + "on port " + s.getPort()) ;
         while (true) {
            // read the timestamp
            timestamp = in.readLine();
            if (timestamp == null) {
              System.out.println("Server closed connection");
              break;
            }
            System.out.println("Daytime : " + timestamp);
         }
      }
      catch (IOException e) { System.out.println(e);}
      finally
      {
          // force the connection closed in case it's open
          try
            {  if (s != null)  s.close(); }
          catch (IOException e2)
            { }
      }
   }
}
```

The code follows our high-level design pretty closely. We first create a socket and then create a stream and tie the two together. Notice that all I/O is done in a try construct so that all I/O problems (socket or stream) are automatically caught as exceptions. In fact, especially notice that the finally clause of the main try/catch/finally uses a nested try to catch the fact that if the connection is already closed so that we can terminate the program gracefully in the null catch statement.

Now that we've mastered the most trivial of the protocols, let's move on to something a little more complicated.

Echo. "Well-known port" 7 on most hosts provides a service called echo. Echo is pretty much a diagnostic service and works as follows (see RFC 862 on the companion CDROM for a fuller description):

1. The client connects to the server on port 7 and proceeds to send data.
2. The server returns everything it receives to the client. This may be done on a character-by-character basis or a line-by-line basis depending on the implementation of the server.

Let's start out our examination of echo by first writing a non-sockets-based version of Echo just to get a feel for what it is that we want to do.

```
public class EchoTest
{
   public static void main (String args[])
   {
      BufferedReader in = new BufferedReader
                          New InputStreamReader(System.in));
      String line;
      while(true)
      {
         line="";
         try
         {
             line = in.readLine();
         }
         catch (IOException e)
         {
            System.err.println(e.getMessage());
         }
      System.out.println(line);
      }
   }
}
```

The program is quite simple and straightforward. First, we define an input stream and connect it to the standard input keyboard (System.in); then we define a string for our only program variable, which will hold the string we read from

the keyboard and print on the Java console. Finally, we put the read and write in a do forever loop. Remember, in Java it is not only considered good form to provide try/catch constructs when doing I/O it is necessaary.

You can execute the program that we created by doing the following, and get similar results:

```
%prompt% javac EchoTest.java
%prompt% java EchoTest
abc                     input...
abc                         ...output
def                     input...
def                         ...output
xyz                     input...
xyz                         ...output
^C
%prompt%
```

Moving EchoTest to Sockets. Taking another step toward proficiency using Java sockets, we modify our echo program to do the following:

1. Read a line from the keyboard.
2. Write it to a socket connected to TCP port 7.
3. Read the reply from the socket connection.
4. Print the line from the socket to the screen.

A socket object is created as follows:

```
Socket s = Socket("localhost", 7);
```

The two arguments to the `Socket` constructor are *hostname* and *port number*. We use "localhost" to keep it simple. The hostname is passed as a `String` variable, typically from the command line and the port number as an `int`.

Here is a simple TCP client written in Java. First, we must create the `EchoClient` class and import all the Java libraries that we will use in our program.

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

public class EchoClient
{
}
```

Now, we must create a function in which we will place a loop similar to the one we created with our Java-only client. This loop must have two objects on which to act—the BufferedReader from the socket from which it will get data and the PrintStream from the socket to which it will write data. We assumed this was standard input and standard output for our Java-only client, but we will not make that assumption here:

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

public class EchoClient
{
    public static void echoclient(BufferedReader  in;
                                  PrintStream out)
                                  throws IOException
    {
    }
}
```

Now, we must get an input stream for the keyboard. For this we'll use another BufferedReader tied to System.in. We will also add the loop here. The loop will first get input from the keyboard using the stream we just created. Then it will write that data directly to the socket.

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

public class EchoClient
{
    public static void echoclient(BufferedReader  in;
                                  PrintStream out)
                                  throws IOException
    {
       kybd = new BufferedReader(
                                 new  InputStreamReader(System.in);
       String line;

       while(true)
       {
          line="";
          // read keyboard and write to the socket
          try
          {
             line = kybd.readLine();
             out.println( line );
          }
          catch (IOException e)
          {
             System.err.println(e.getMessage());
          }
       }
    }
}
```

To finish up, we now read the activity on the socket and stick it on the screen by writing to the Java console using the System object.

```
public class EchoClient
{
```


```
   public static void echoclient(BufferedReader in,
                                 PrintStream out)
                                 Throws IOException
   {
      // make a stream for the keyboard
      BufferedReader kybd = new BufferedReader(
                               new InputStreamReader(
                                    System.in));
      String line;   //for reading into
      while(true)
      {
         line="";
         // read keyboard and write to TCP socket
         try
         {
            line = kybd.readLine();
            out.println( line );
         }
         catch (IOException e)
         {
            System.err.println(e.getMessage());
         }
         // read TCP socket and write to java console
         try
         {
            line = sin.readLine();
            System.out.println(line);
         }
         catch (IOException e)
         {
            System.err.println(e.getMessage());
         }
      }
   }
}
```

Finally, we can create our main application. In our main application, we will create the socket first and then get a BufferedReader and a PrintStream based on it. This enables us to read and write to the socket easily, as well as pass it on to the function we created earlier. Once we are finished, we must close the connection to the socket.

As we will discuss later, too many open connections are a system liability. If a connection is not in use, but is still open, other applications may not be able to connect to the port to which you are connected.

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

public class EchoClient
{
   public static void echoclient(BufferedReader in,
                                  PrintStream    out)
                                  throws IOException
   {
      // make a stream for reading the keyboard
      BufferedReader kybd = new BufferedReader(
                                new InputStreamReader(
                                    System.in));
      String line;
      while(true)
      {
         line="";
         // read keyboard and write to TCP socket
         try
         {
             line = kybd.readLine();
             out.println(line);
         }
         catch(IOException e)
         {
             System.err.println(e.getMessage());
         }
         // read TCP socket and write to console
         try
         {
             line = in.readLine();
             System.out.println(line);
         }
         catch(IOException e)
         {
             System.err.println(e.getMessage());
         }
      }
   }

   public static void main(String[] args )
   {
      Socket s = null;
      try
      {
         // Create a socket to communicate with "echo"
         // on the specified host
         s = new Socket(args[0], 7);
```


```
            // Create streams for reading and writing
            // lines of text from and to this socket.
            BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                s.getInputStream()));
            PrintStream out = new(
                                PrintStream(s.getOutputStream());
            // Tell the user that we've connected
            System.out.println("Connected to " +
                                s.getInetAddress() + ":" + s.getPort());
            echoclient(in, out);
         }
         catch (IOException e)
         {
            System.err.println(e);
         }
         // Always be sure the socket gets closed
         finally
         {
            try
            {
               if(s != null)  s.close();
            }
            catch (IOException exc)
            { ; /* terminate gracefully */}
         }
      }
}
```

When we execute our program, we send a message to the Echo socket, read whatever information comes back on the socket, and then print it. Because the echo socket merely takes whatever input it gets and bounces it right back to the port, what we get in return on the socket is exactly what we sent. The output is displayed next. If you need to connect to another host, substitute its name for localhost.

```
%prompt% java EchoClient localhost
Connected to localhost/127.0.0.1:7
abc        request...
abc            ...reply
xyz        request...
xyz            ...reply
^C
```

This service (and most others) can be tested using the Telnet client that is available as an application with most TCP/IP stacks. In this case, the Telnet program acts in the same manner as our client, sending information to the port and reading whatever it gets back.

```
%prompt% telnet localhost 7
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
abc        request...
abc             ...reply
^C
xyz        request...
xyz             ...reply
^]    control-right-bracket
telnet> quit
Connection closed.
```

URL and URL Connection

Before we leave the topic of using sockets to connect existing Internet servers, let's look at using some of the more common and popular services provided on the Internet. We need to examine a couple of other members of *java.net*: URL and URL Connection.

A Uniform Resource Locator (URL) is a string that identifies a resource on the Internet. RFC 1738 gives an in-depth description of everything you would ever want to know about URLs. Table 3-3 is a brief description of the various things that make up a URL.

Table 3–3 Makeup of a URL

Protocol	An identifier (usually an acronym) that specifies the protocol to use to access the resource
Host name	The name of the host or domain where the resource is located (*www.binghamton.edu*, localhost)
Port number	The TCP/IP port number that the service is being provided on
Filename	Path- and filename of resource
Reference	#anchorname

The URL class gives us the ability to construct URL objects and a number of "getter" methods that let us extract the various parts of a URL. From a networking standpoint, the methods of getContent(), openConnection(), and openStream() provide us with some very useful tools that we can use to interface with a number of protocol servers.

To retrieve a file from a Web server, all we really need to know is its URL:

```
Class GetURL
{
   try
   {
```


```
    String host = "watson2.cs.binghamton.edu";
    String file = "~steflik/index.html";
    String line;
    BufferedReader in;
    URL  u = new URL("http://"+host+"/"+file);
    Object content = u.getContent();
    System.out.println("class: " + content.getClass());
    System.out.println("content: " + content.toString());
    In = new BufferedReader
             (new InputStreamReader(u.openStream()));
    while ((line = in.readLine() != null)
    {
      System.out.println(line);
    }
  }
  catch (MalformedURLException e) { e.printStackTrace();}
  catch (IOException e) {e.printStackTrace();}
}
```

All we had to do was create a URL object and then use the openStream() method to create an InputStream and eventually a BufferedReader that we can use to retrieve the file. At this point all that is needed is a loop to read the lines out of the file.

This technique can be exploited for doing things like populating selection lists in an applet-based shopping cart application with data from a set of pricing files kept on the Web server. This technique can also be used to run scripts stored on a Web server.

Summary of Sockets

We have shown you what, in the most basic sense, sockets are and how they are used in Java to build client applications that communicate, using well-defined protocols with standards-based (developed using the RFC process) servers. The subsequent sections in this chapter build on this material and show you how to create an entire client/server system using only sockets. The rest of this book showcases several other Java communication technologies that use sockets as their underlying mechanism to transfer data across networks. In the large of it, applications use protocols to direct the way they talk to one another and protocols use sockets as their network interface.

Client/Server Methodology

In the previous section we developed client applications for servers that already exist. This isn't the way that we would necessarily approach developing a sockets-based client/server application. In the next few pages we will examine a client/server application for an Internet-based pizza ordering/delivery service that will be made up of a client (that pizza lovers around the community can install on the

home computers to order a pizza), a server (running at the store), and a protocol that directs the information exchange between the client and the server.

Suppose that you are at home with your cronies watching the Super Bowl, and, as luck would have it, the Washington Redskins are playing. As invariably happens, you've run out of nachos and dip before half time, so you decide to replenish the nutrition supply by ordering a pizza. Today, when you want to order that pizza, you pick up the phone and call your favorite pizzaria to request a delivery.

A few years ago, a small start-up company in the Silicon Valley called the Santa Cruz Operation (SCO) developed an Internet pizza-ordering application. By today's standards, it was quite low-tech, based solely on HTML forms and requiring someone to read the information manually on the other end via e-mail. The nifty thing about this Internet Pizza Hut was the idea that you could simply use your computer to communicate with a faraway place and get a pizza. In this sense, SCO was pretty well ahead of the game—they were among the first to genuinely use the Internet, not the corporate intranet, to conduct business with remote users.

In this section, we will develop our own pizza client/server system as an ultra-hip high-tech alternative to the telephone and publish it to the world. This time, however, we will use Java and implement our `PizzaServer` using sockets.

The Pizza Order Protocol (TPOP)

To design the protocol we need to examine what information must be passed from the client to the server and vice versa. If the user interface for our client application is as shown in Figure 3-1 we can readily see that to constitute an order we need to send the name, address, phone number, pizza size (small, medium, or large) , and which topping (Veggies, Meat, or California) is to be added to a standard cheese pizza.

The protocol that is required to place an order is pretty simple, as shown in Table 3-4.

Let us further decide that, since we're in this early part of design, all data exchanged between the client and the server is to be as plain old text strings (in the true tradition of the Internet), each of which is to be delimited by the "|" character.

The next decision we need to make is which component we will develop first: the client or the server. If we choose to develop the client first, we won't be able to test it until we develop the server and then end up with the possibility of having to use two untested pieces of software to test each other. Realizing the possible disaster that can occur if this avenue is followed, let's think about developing the server first. If the server is running, we can always test it using our Telnet client. To do this, all we do is start up our Telnet client, connect to port 8205 of the server,

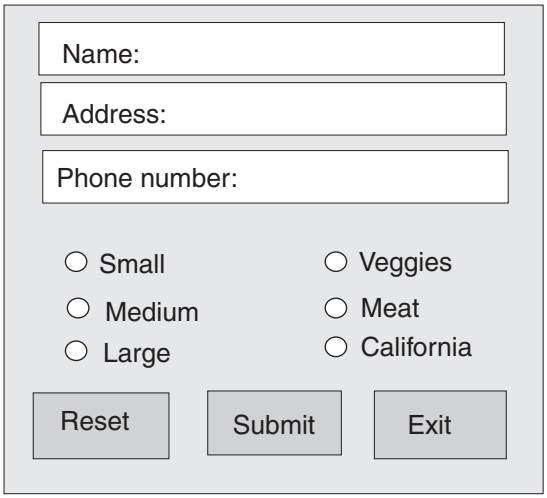

Figure 3–1 A sample GUI for the PizzaTool.

Table 3–4 TPOP

Client	Server
	Start server listening on port 8205
Connect to port 8205 of the server	
	Accept the connection and spawn a thread to handle the connection data
Send the order information and then wait for the price to display	
	Receive the order, print it out, calculate the price, return price to client application, and break the connection
Display the price	

type in the data separated by "|" characters, and press Enter. The server will process the data, send back the price information, and, close the connection. This approach helps set us up for success rather than failure.

The TPOP Server

Server Methodology

For every client there must somewhere be a server. In an attempt to make server creation as simple as possible, Java provides a Server Socket class as part of *java.net*. Server Sockets, once created, listen on their assigned port for client

connection requests. As requests are received, they are queued up in the Server Socket. The Server Socket accepts the connection request; as part of this acceptance the Server Socket creates a new socket, connects it to the client, and disconnects the connection on the Server Socket port, leaving it open for more connection requests. The client and server now talk back and forth on the new socket connection, and the server listens for connection requests on the Server Socket.

This all sounds pretty simple, but we haven't mentioned anything about threads yet. One of the basic ideas of client/server methodology is that one server should service as many clients as possible. To do this there must be something in the recipe that provides parallelism. That something is threads. The Thread class provides Java with a consistent, operating system neutral way of using the threading capabilities of the host operating system.

Java threads, sockets, and AWT components are similar in that the classes provided are really interfaces to the threads, sockets, and GUI widgets supplied by the operating system that is hosting the Java virtual machine. This means that if you are on Windows, you are really interacting with the TCP/IP protocol stack provided by *winsock.dll*; if you are on a UNIX platform, you are most likely using Berkley sockets. If you are on a Sun Solaris, you are using the threading provided by the Solaris operating system. If you are on Windows 98 using AWT widgets, you are really using the widgets provided by Windows. Used this way by Java, these components are known as peer components or objects. The adding of the Swing components to Java 1.1 starts to get away from this by providing 100% Java GUI components.

A typical TCP application opens a "well-known" port to receive connection requests, and then it spawns a child process or a separate thread of execution to perform the requested service. This ensures that the server is always ready for more invocations. A single-threaded server must poll the sockets constantly. When it detects activity, it must spawn a new process to handle the incoming request. Our multithreaded server can simply wait for information on a socket and spawn a thread to handle incoming requests.

The `PizzaServer` that we will implement will hang on port 8205 and wait for information. When the client sends its bar-delimited request, the server will spawn a thread to handle the request. The thread reads the information, processes it, and sends a reply.

Setting Up the Server

We must create the `PizzaServer` object itself. The `PizzaServer` is a stand-alone Java application with its own application main (on the accompanying CD, two

versions of the server are provided—one with a GUI interface and one without). We must also create a `PizzaThread` that inherits from the Java `Thread` class. This threaded object will be created every time we detect activity on the port. As we discussed in our Chapter 1 section on threads, it is one of two ways we could have implemented the server object. We leave the other threaded version as an exercise to the reader.

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

public class PizzaServer
{
    public static void main(String args[])
    {
    }

    // threaded pizza!
    class PizzaThread extends Thread
    {
    }
```

Initializing the Server Socket

Inside the main program, we must create a `ServerSocket`. The `ServerSocket` is a Java type whose sole purpose is to enable you to wait on a socket for activity. Initialize it by specifying the port on which you want to wait.

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

public class PizzaServer
{
    public static void main(String args[])
    {
       // initialize the network connection
       try
       {
         ServerSocket serverSocket = new ServerSocket(8205);
       }
       catch(Exception exc)
       {
         System.out.println("Error! - " + exc.toString());
       }
    }
}

// threaded pizza!
```

```
class PizzaThread extends Thread
{
}
```

Creating the Thread

The `PizzaThread` object will accept one variable, the `incoming` socket from which it gathers information. We need to specify this here because the main server program has already grabbed hold of the socket, and we don't want to do so twice. We merely pass the socket obtained by the main program on to the thread. We will also implement the `run` method for the thread.

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

public class PizzaServer
{
   public static void main(
      String args[]
   )
   {
      // initialize the network connection
      try
      {
         ServerSocket serverSocket = new ServerSocket(8205);
      }
      catch(Exception exc)
      {
         System.out.println("Error! - " + exc.toString());
      }
   }
}

// threaded pizza!
class PizzaThread extends Thread
{
    // the socket we are writing to
    Socket incoming;

    PizzaThread(    Socket incoming )
    {
       this.incoming = incoming;
    }

    // run method implemented by Thread class
    public void run()
    {
    }
}
```

Detecting Information and Starting the Thread

Now, we must wait on the thread until activity occurs. Once we detect some semblance of information coming across the socket, we must spawn a thread automatically and let the thread get and process the information. Our main program merely delegates activity to others.

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

public class PizzaServer
{
    public static void main(String args[])
    {
       // initialize the network connection
       try
       {
           ServerSocket serverSocket = new ServerSocket(8205);

           // now sit in an infinite loop until we get something
           while(true)
           {
             // accept the message
             Socket incoming = serverSocket.accept();

             // spawn a thread to handle the request
             PizzaThread pt = new PizzaThread(incoming);
             pt.start();
           }
       }
       catch(Exception exc)
       {
           System.out.println("Error! - " + exc.toString());
       }
    }
}

// threaded pizza!
class PizzaThread extends Thread
{
    // the socket we are writing to
    Socket incoming;

    PizzaThread(Socket incoming)
    {
        this.incoming = incoming;
    }

    // run method implemented by Thread class
    public void run()
```

```
    {
    }
}
```

Notice also how we must call the `start` method explicitly on the thread. As we discussed in the Threads section of Chapter 1, if a class inherits from the Java `Thread` class, the thread must be started from outside the class.

Gathering Information

Once the thread is running, it needs to go to the socket and get information. To do so, we must obtain input and output streams to read and write to/from the socket. Remember that the socket is merely a construct. In order to get information from it, it must be abstracted into an input/output mechanism. We will then be able to read and write to the socket. As we will discuss in our client section, the data we are going to receive is in a bar-delimited format. We must use a `StringTokenizer` object to extract the information from the message.

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

public class PizzaServer
{
    public static void main(String args[])
    {
        // initialize the network connection
        try
        {
            ServerSocket serverSocket = new ServerSocket(8205);

            // now sit in an infinite loop until
            // we get something
            while(true)
            {
                // accept the message
                Socket incoming = serverSocket.accept();

                // spawn a thread to handle the request
                PizzaThread pt = new PizzaThread(incoming);
                pt.start();
            }
        }
        catch(Exception exc)
        {
            System.out.println("Error! - " + exc.toString());
        }
    }
}
```


```
// threaded pizza!
class PizzaThread extends Thread
{
   // the socket we are writing to
   Socket incoming;

   PizzaThread(Socket incoming)
   {
      this.incoming = incoming;
   }

   // run method implemented by Thread class
   public void run()
   {
      try
      {
         // get input from socket
         DataInputStream in =
            new DataInputStream(incoming.getInputStream());

         // get output to socket
         PrintStream out =
            new PrintStream(incoming.getOutputStream());

         // now get input from the server until it closes the
         // connection
         boolean finished = false;
         while(!finished)
         {
            String newOrder = in.readLine();

            // convert to a readable format
            try
            {
               StringTokenizer stk =
                  new StringTokenizer(newOrder, "|");
               String name = stk.nextToken();
               String address = stk.nextToken();
               String phone = stk.nextToken();
               int size =
                  Integer.valueOf(stk.nextToken()).intValue();
               int toppings =
                  Integer.valueOf(stk.nextToken()).intValue();

               // no exception was thrown so calculate total
               int total = (size * 5) + (toppings * 1);

               // send the result back to the client
               out.println("$" + total + ".00");

               // put our result on the screen
               System.out.println("pizza for " + name +
                  " was " + totalString);
```

```
            }
            catch(NoSuchElementException exc)
            {
               finished = true;
            }
         }
      }
      catch(Exception exc)
      {
         System.out.println("Error! - " + exc.toString());
      }
      // close the connection
      try
      {
         incoming.close();
      }
      catch(Exception exc)
      {
         System.out.println("Error! - " + exc.toString());
      }
   }
}
```

Note in particular the two lines we actually use for reading information from the socket and sending information back:

```
String newOrder = in.readLine();
// send the result back to the client
out.println("$" + total + ".00");
```

These two lines have the same syntax as they would if they were reading and writing a file. In fact, as we discussed in Chapter 1's input/output section, to the programmer a socket is nothing more than a file. We are able to use streams, read and write information, and save sockets just as we would files. This is an important concept to grasp because the security restrictions that apply to sockets also apply to files. We will discuss security in greater detail in Chapter 13, "Java and Security."

The TPOP Client

Clients are the end user interface to an application and end up being responsible mainly for collecting user input and sending it to the server. Servers are the recipients of that information. Think of a client approaching your restaurant with a pocket full of money and you the owner, as the server, gladly accepting that money for your product and services. In this section we begin our discussion of client/server programming by developing an application that transmits information across a network connection to another program.

Developing Clients for Servers

The `PizzaTool` we are about to create is a stand-alone Java application and will have a fancy GUI interface that you can design yourself. Our GUI code's framework looks something like this:

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

public class PizzaTool extends Frame
{
    // AWT Components
    . . . skip these for now . . .

    PizzaTool()
    {
       // initialize the application frame

       // create the GUIs
    }

    public boolean action(
          Event evt,
          Object obj
    )
    {
       if(evt.target == sendButton)
       {
       }

       return true;
    }

    public static void main(
      String args[]
    )
    {
       PizzaTool pizza = new PizzaTool();
       pizza.show();
    }
}
```

When displayed, our pizza tool GUI will look something like the one shown in Figure 3-1.

We need to modify this working client to send its information over the network to the other end. To do so, we must create a socket in our application's constructor and initialize it as we did earlier. We will use port number 8205 in this application.

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

NOTE: As will be our practice throughout this book, we show you the completed GUI rather than showing the code development process for it. There are several GUI builders on the market, and we hope you will choose one to assist you. If you are a neophyte at Java, we recommend using a text editor and Sun's JDK (Java Development Kit) until you become proficient at Java. GUI builders like Visual Café, J++, and JBuilder are great tools and can really increase productivity; the problem is that they really hide a lot (especially in building the user interface) from you. In some cases, the code produced by the GUI builders is not necessarily good code, but it is code that will work.

```
public class PizzaTool extends Frame
                        implements ActionListener
{
    // AWT Components
    . . . skip these for now . . .
    // network components
    Socket socket;
    DataInputStream inStream;
    PrintStream outStream;
    PizzaTool()
    {
       // initialize the application frame
       // create the GUIs
       // define Exit button handler
       // define Reset button handle
    }
    public void actionPerformed(ActionEvent e)
    {
       // handle the Submit button here, build and send the order
    }
    public static void main(String args[])
    {
       // use the constructor to build the GUI
       PizzaTool pizza = new PizzaTool();
       // show the GUI and wait for an Action Event
       pizza.show();
    }
}
```

Inside the `actionPerformed` method, we need to send the information we gather from our GUI back to the server. The server then makes a calculation and sends us the total for the order. First, we must send information across the socket using the `outStream` variable we derived from the socket. Then, just as we did earlier, we

must turn around and read information from the same socket using the `inStream` variable.

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

public class PizzaTool extends Frame
                       implements ActionListener
{
    // AWT Components
    . . . skip these for now . . .

    // network components
    Socket socket;
    DataInputStream inStream;
    PrintStream outStream;

    PizzaTool()
    {
       // initialize the application frame

       // create the GUIs

       // define Exit button handler

       // define Reset button handler
       resetButton = new Button("Reset Order");
       resetButton.setBounds(160,270,140,60);
       add(resetButton);
       resetButton.addActionListener(
          new ActionListener()
          {
             public void actionPerformed(ActionEvent e)
             {
                instructionField.setText("Select Pizza");
                nameField.setText("");
                addressField.setText("");
                phoneField.setText("");
             }
          }
       );
    }

    public void actionPerformed(ActionEvent e)
    {
        // handle the Submit button here, build and send the order
        // create the socket and attach input and output streams
        try
        {  // open the socket to the remote host
           socket = new Socket("localhost", 8205);
           in = new BufferedReader(
```

```
                    new InputStreamReader(socket.getInputStream()));
               outStream = new PrintStream(socket.getOutputStream());
            }
            catch (Exception e)
            {
               System.out.println(IO Exception: " + e.toString());
            }
            // Send the order to the server
            instructionField.setText("Sending order");
            try
            {
               outStream.println(
                       nameField.getText() + "|" +
                       addressField.getText() + "|" +
                       phoneField.getText() + "|" +
                       size + "|" +
                       toppings);
            }
            catch (Exception e)
            {
               System.out.println("Error: " + e.toString());
            }
            // read the price from the server
            String totalString = new String();
            try
            {
               totalString = inStream.readLine();
            }
            catch (Exception e)
            {
               System.out.println("Error: " + e.toString());
            }
        }
        public static void main(String args[])
        {
           // use the constructor to build the GUI
           PizzaTool pizza = new PizzaTool();
           // show the GUI and wait for an Action Event
           pizza.show();
        }
}
```

Please check out the bold, italicized text that defined the Reset button and its event handler. This looks a little strange but really isn't; what you are looking at is an anonymous inner class being used as the event handler. With the new event model that came about with JDK 1.1 came some improved event handling. Using an anonymous inner class, the event handler can be kept right with the code

(this aids maintainability) and eliminates the need for large if/then/else structures for decoding what caused the event. This makes the code run considerably faster.

Notice also how we send information to the server. We have created our own protocol and message format to use to send the three important customer fields, as well as the kind of pizza ordered, directly to the pizza server. The format is delimited by the bar sign ("|") and, as we will see in a moment, is interpreted on the server end.

```
outStream.println(
    nameField.getText() + "|" +
    addressField.getText() + "|" +
    phoneField.getText() + "|" +
    size + "|" +
    toppings);
```

Once complete, our application then is able to publish the information it received from the server.

NOTE: In order to conserve paper (save some trees), we have not shown you the entire code listing for both the GUI and the network portion of our application. As always, a full, working version of this application can be found on the CD-ROM that accompanies this book.

Socket programming is at the heart of everything we discuss in this book. Every communication technology involved with computers uses sockets in some fashion. Often, having control over the format and length of messages between clients and servers is of great importance. We could just as easily have created our pizza application using a mechanism found in other parts of this book. However, by using sockets, we had full control over how the communication (protocol) is implemented.

Clients and Servers in Short

So far we have implemented an application for which we know what is on both ends. This form of point-to-point communication is one way to create a networked application. We created a message, located the destination for the message, and shipped it off. While reliable, point-to-point communication is important, we also want to be able to form a message and broadcast it. In so doing, anyone anywhere can grab the message and act on it. This form of broadcast communication can also be accomplished using Java sockets and is discussed in the next section.

UDP Client

We have spoken so far about TCP communication, which we have mentioned is a point-to-point, reliable protocol. Well, what makes an unreliable protocol? An unreliable protocol is one in which you send a chunk of information, and if it gets lost along the way, nobody really minds. TCP provides an infrastructure that ensures a communication is sent and arrives safely. Another protocol, User Datagram Protocol (UDP), is a "spit in the wind" protocol. One day, you wake up, spit into the wind, and hope it will land somewhere. Likewise, with datagrams you can easily form a message, send it, and hope it gets to the other end. There are no guarantees that it will ever arrive, so be careful when choosing to use a UDP socket over a TCP-based socket for your application.

Datagrams

In the last chapter, we referred to datagrams as letters that we send to a mailbox. In fact, a datagram is a chunk of memory, not unlike a letter—a chunk of paper into which we put information and send off to a mailbox. Just as with the U.S. Postal Service, there is absolutely no guarantee that the letter will ever arrive at its destination.

Here's a sample "receive buffer" datagram:

```
DatagramPacket packet = new DatagramPacket(buf, 256);
```

You must give the constructor the name of a byte or character array to receive the data and the length of the buffer in bytes or characters. You get data as follows:

```
socket.receive(packet);
```

where socket is created as follows:

```
socket = new DatagramSocket();
```

The `DatagramSocket` class is an endpoint (mailbox) for UDP communication. Like the `Socket` class (which uses TCP), there is no need for a programmer to specify the transport-level protocol to use.

After a datagram is received, you can find out where it came from as follows:

```
address = packet.getAddress();
port = packet.getPort();
```

and you can return a reply as follows:

```
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```

This datagram will go out the same UDP port (akin to a "mailbox"), to the other process-receiving datagrams on that UDP port number. A UDP server can specify its service port number in its constructor, in this case port number 31543.

```
socket = new DatagramSocket(31543);
```

NOTE: Datagrams are sort of like that old "I Love Lucy" episode in which Lucy and Ethel go to work in a candy factory. As they stand in front of a conveyor belt, little candies begin to flow out. Lucy and Ethel are able to wrap and package the candies as they come out. Soon, their boss speeds up the belt, and the candies begin to flow out really fast; Lucy and Ethel are unable to keep up. Similarly, datagrams happen along the port and are picked up by receiver programs that happen to be listening. Unlike Lucy and Ethel, however, if you miss one, nothing bad will happen.

Creating a UDP Sender

To pay homage to Lucy and Ethel in our own bizarre, twisted way, let's create a cookie factory! In our factory, we will be able to build chocolate chip cookies and specify the number of chips we want in each one. Then we will send them along the conveyor belt to be packaged and shipped off to some Java engineer turned writer who is in desperate need of a Scooby Snack.

Real-world implementations of broadcast communication include stock tickers that constantly publish stock quotes for NASDAQ or the New York Stock Exchange. By simply plugging your receiver into the port, you can grab that information and do something with it (like displaying it as a ticker tape message across the bottom of your screen). Modifying our sample program to similarly broadcast and grab information is quite simple.

To begin our sender program, we must create a Java application for our `CookieBakery`. The application will have a simple GUI in which you can specify the number of chips in the cookie using a slider and then simply press a button to send the cookie to the conveyor belt.

The GUI framework looks like this:

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

public class CookieBakery extends Frame
{
    // AWT components

    CookieBakery() //constructor
    {
      // initialize the application frame
      // build the GUI and event handlers

      sendButton = new Button("Send Cookie");
      sendButton.setBounds(10,270,290,60);
      add(sendButton);
      sendButton.addActionListener(
```

```
        new ActionListener()
        {
          public void actionPerformed(ActionEvent e)
          {
              // determine the number of chips
              int numChips = chipsScrollbar.getValue();
              String messageChips = numChips + " chips";
              // build the message and send it
                    // display final result
                    instructionField.setText(
                               "Sent Cookie with " +
                                numChips + " chips!");
          }
    }
    // all events handled by inner classes, this is required
    public void actionPerformed(Event e){;)
    public static void main(String args[])
    {
        CookieBakery cookies = new CookieBakery();
        cookies.show();
    }
}
```

The GUI itself will resemble that shown in Figure 3-2 with a slider to select the number of chips and a button to press so that you can "bake" it.

Formatting a UDP Packet. In order to send a packet to the server, we must create and format one. Packets are created using buffers and contain an array of bytes. Therefore, any string message that you wish to send must be converted to an array of bytes. We will do this in a moment. Also, we need to define and obtain

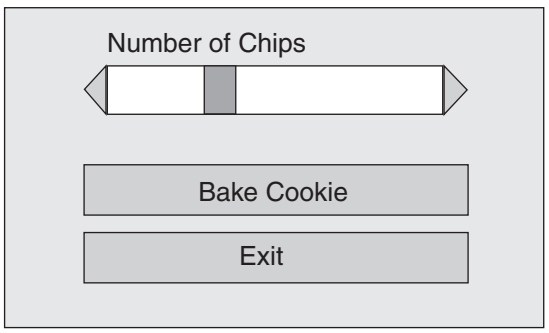

Figure 3–2 Sample GUI for the CookieBakery.

the Internet address of the machine on which this application runs. UDP requires it as part of its protocol.

```
import java.awt.*;
import java.net.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class CookieBakery extends Frame
{
    // AWT components

    CookieBakery() //constructor
    {
      // initialize the application frame
      // build the GUI and event handlers

      sendButton = new Button("Send Cookie");
      sendButton.setBounds(10,270,290,60);
      add(sendButton);
      sendButton.addActionListener(
         new ActionListener()
         {
           public void actionPerformed(ActionEvent e)
           {
               // determine the number of chips
               int numChips = chipsScrollbar.getValue();
               String messageChips = numChips + " chips";

               // convert the chip message to byte form
               int msgLength = messageChips.length();
               byte[] message = new byte[msgLength];
               message = messageChips.getBytes();

               // send a message
               try
               {
                 // format the cookie into a UDP packet
                 instructionField.setText(
                             "Sending Cookie...");
                 DatagramPacket packet = new DatagramPacket(
                          message, msgLength,
                          internetAddress, 8505);

                 // send the packet to the server
                      DatagramSocket socket = new
                                              DatagramSocket();
                      socket.send(packet);
               }
                  catch(Exception exc)
               {
                  System.out.println("Error! - " +
                                          exc.toString());
```

```
                }
                // display final result
                instructionField.setText(
                        "Sent Cookie with " +
                         numChips + " chips!");
            }
        }
        public void actionPerformed(Event e){;}
        public static void main(String args[])
        {
             CookieBakery cookies = new CookieBakery();
             cookies.show();
        }
    }
}
```

Sending the Packet to the Server. In order to send the cookie to the conveyor belt, we must create a `DatagramSocket`. Then we can send the packet we just created using the `send` routine.

```
import java.awt.*;
import java.net.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class CookieBakery extends Frame
{
    // AWT components

    CookieBakery() //constructor
    {
      // initialize the application frame
      // build the GUI and event handlers

      sendButton = new Button("Send Cookie");
      sendButton.setBounds(10,270,290,60);
      add(sendButton);
      sendButton.addActionListener(
         new ActionListener()
         {
           public void actionPerformed(ActionEvent e)
           {
               // determine the number of chips
               int numChips = chipsScrollbar.getValue();
               String messageChips = numChips + " chips";

               // convert the chip message to byte form
               int msgLength = messageChips.length();
               byte[] message = new byte[msgLength];
               message = messageChips.getBytes();
```


```
                // send a message
                try
                {
                   // format the cookie into a UDP packet
                   instructionField.setText(
                                "Sending Cookie...");
                   DatagramPacket packet = new DatagramPacket(
                            message, msgLength,
                            internetAddress, 8505);
                   // send the packet to the server
                   DatagramSocket socket =new DatagramSocket();
                   Socket.send(packet);
                }
                catch(Exception exc)
                {
                   System.out.println("Error! - " +
                                      exc.toString());
                }
                // display final result
                instructionField.setText(
                              "Sent Cookie with " +
                               numChips + " chips!");
             }
          }
          public void actionPerformed(Event e){;}
          public static void main(String args[])
          {
             CookieBakery cookies = new CookieBakery();
             cookies.show();
          }
       }
}
```

Now that we have created an application that sends a message containing "xx chips" to a port, we need something on the other end to receive and decode the message into something useful. After all, we don't want to waste our delicious chocolate chip cookies!

Creating a UDP Receiver. The UDP Receiver we will create will listen in on a port and wait for cookies. When it gets one, our `CookieMonster` will let us know by printing a "Yummy, tastes good" message. As with our `CookieBakery`, the `CookieMonster` will listen in on port 8505, a totally random selection. To start our `CookieMonster`, first we must create the `CookieMonster` object with its own application main containing the packet that we will read and the socket from which we will get it. Note that we are importing the `java.net.*` package once again.

NOTE: We find throughout this book that servers, in this case a receiver, must be applications, whereas clients very easily can be applets as well. The reason is that Java's security mechanism will not allow a downloaded applet to have unlimited access to a port on the machine to which it is downloaded. Because of the Java security model, you are prevented from developing downloadable servers. This may change with the introduction of browsers that are able to change those security restrictions.

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

public class CookieMonster
{
    public static void main(
        String args[]
    )
    {
        // our socket
        DatagramSocket socket = null;

        // our packet
        DatagramPacket packet = null;
    }
}
```

Now we must create and initialize the packet that we will receive. Note that we have to specify a buffer into which the packet will read the message. A packet by itself is composed of four elements. The first is shown in the following code.

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

public class CookieMonster
{
    public static void main(
        String args[]
    )
    {
       // our socket
       DatagramSocket socket = null;

       // our packet
       DatagramPacket packet = null;

          // create a receive buffer
          byte[] buffer = new byte[1024];

          // create a packet to receive the buffer
          packet = new DatagramPacket(buffer, buffer.length);
    }
}
```

Once our packet is put together, we need to sit on a socket and wait for someone to fill it with information. We use the `DatagramSocket`'s `receive` routine to hang on a UDP port and get information. We must pass the packet to the sockets `receive` method so that the packet knows where to put the information it gets.

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

public class CookieMonster
{
   public static void main(
     String args[]
   )
   {
      // our socket
      DatagramSocket socket = null;

      // our packet
      DatagramPacket packet = null;

      // create a receive buffer
      byte[] buffer = new byte[1024];

      // create a packet to receive the buffer
      packet = new DatagramPacket(buffer, buffer.length);

      // now create a socket to listen in
      try
      {
          socket = new DatagramSocket(8505);
      }
      catch(Exception exc)
      {
          System.out.println("Error! - " + exc.toString());
      }

      // now sit in an infinite loop and eat cookies!
      while(true)
      {
         // sit around and wait for a new packet
         try
         {
             socket.receive(packet);
         }
         catch(Exception exc)
         {
              System.out.println("Error! - " + exc.toString());
         }
      }
   }
}
```

So now we have a cookie in our hands, and we have to somehow eat it. To do so, we must first extract the cookie from the packet by retrieving the packet's buffer.

Because we specified the buffer size when we created the packet, the `CookieMonster` waits until the buffer is filled before it returns the packet. This means that if the packets on the sending end are smaller than the packets we are reading here, we will end up with a packet, plus a little bit of the packet that comes down the pike afterwards, causing havoc in our messaging system. If our buffer is too large on the sending end, we will receive only a little bit of the message. It is important that you synchronize both the receiver and the sender so that they receive and send the same size buffer.

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

public class CookieMonster
{
    public static void main(
        String args[]
    )
    {
      // our socket
      DatagramSocket socket = null;

      // our packet
      DatagramPacket packet = null;

      // create a receive buffer
      byte[] buffer = new byte[1024];

      // create a packet to receive the buffer
      packet = new DatagramPacket(buffer, buffer.length);

      // now create a socket to listen in
      try
      {
          socket = new DatagramSocket(8505);
      }
          catch(Exception exc)
      {
          System.out.println("Error! - " + exc.toString());
      }

      // now sit in an infinite loop and eat cookies!
      while(true)
```


```
        {
            // sit around and wait for a new packet
            try
            {
                socket.receive(packet);
            }
            catch(Exception exc)
            {
                System.out.println("Error! - " + exc.toString());
            }

            // extract the cookie
            String cookieString = new String(buffer, 0, 0,
                    packet.getLength());

            // now show what we got!
            System.out.println("Yummy!  Got a cookie with " +
                    cookieString);
        }
    }
}
```

Now that we have learned how to create point-to-point and broadcast communication mechanisms, let's apply our knowledge to implement our featured application. In this real-world scenario, we must create a mechanism that enables a client to change its state and to send that information to a server to be stored and retrieved at a later date. To develop such an application, we need a point-to-point protocol because reliability is of the utmost premium. After all, we don't want to schedule an appointment and not know if it actually got on our calendar.

Featured Application

As we discussed in Chapter 1, "Advanced Java," we will reimplement the same "featured application" in this chapter and in each of the next four chapters. We hope that this gives you an insight into the advantages and disadvantages of each of the major communication alternatives that we present in this book. Our socket implementation needs to be preceded by a discussion on how we plan to implement messaging between the client and the server. Once that is complete, we can implement the client and the server to exchange information in that format.

Messaging Format

Our messaging format must incorporate the two major elements contained in our notion of an appointment—the time of the appointment and the reason for the appointment. Therefore, we will create a message format akin to the Pizza

Tool's message. In the Pizza Tool we implemented a few sections ago, we delimited our message with the bar symbol ("|"). Once again, we will use the bar symbol to separate the time and reason in our message from the client to the server.

From the server to the client, we need a slightly similar but more robust format. When the server sends information to the client, we will need to string a variable number of bar-delimited appointments together. The client can then use the `StringTokenizer` object to extract the information it needs.

But, the client cannot accept messages without asking for them first. Therefore, we need a header to the message. When we schedule an appointment (i.e., send a message from the client to the server), we precede the message by the word "store." When we merely prompt the server to send the client a message (i.e., the client sends a message to the server telling it to go ahead and reply), we precede the message with the word "retrieve."

Therefore, our message will be in one of the following two formats:

```
store|Take Fleagle to dentist|1
retrieve
```

The retrieve message prompts the server to send a message back with appointments strung together but delimited by the bar symbol.

Client

Because implementing the client for the featured application is quite similar to the Pizza Tool's client, the code we are about to produce will look remarkably similar to the code for the Pizza Tool. In order to plug our featured application socket implementation directly into the Calendar Manager, we must implement the `NetworkModule` that we declared in Chapter 1.

```
public class NetworkModule
{
    public void scheduleAppointment(String reason, int time);

    public Vector getAppointments();

    public void initNetwork();

    public void shutdownNetwork();
}
```

Specifically, we need to implement the `scheduleAppointments` and `getAppointments` methods. We will also have to create and implement a constructor to open and establish the socket connection. We will first implement the constructor. The code is basically cut and pasted directly from the Pizza Tool:

```
import java.awt.*;
import java.util.*;
```


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

public class NetworkModule
{
   // network components
   Socket socket;
   DataInputStream inStream;
   PrintStream outStream;

   NetworkModule()
   {
      try
      {
         socket = new Socket("localhost", 8205);
         inStream = new BufferedReader(
                    new InputStreamReader(
                    socket.getInputStream()));
         outStream = new
                    PrintStream(socket.getOutputStream());
      }
      catch(Exception exc)
      {
         System.out.println("Error! - " + exc.toString());
      }
   }

   public void scheduleAppointment(
                  String appointmentReason,
                  int appointmentTime)
   {
   }

   public Vector getAppointments()
   {
   }

   public void initNetwork()
   {
   }

   public void shutdownNetwork()
   {
   }
}
```

Now we must implement the `scheduleAppointment` method that goes to the server with a formatted message containing the new appointment. Notice how we put together the message so that it conforms to the messaging format we just agreed upon.

```
public void scheduleAppointment(
```

```
          String appointmentReason,
          int appointmentTime )
{
   try
   {
       outStream.println(
            "store|" +
            appointmentReason + "|" +
            appointmentTime + "|");
   }
   catch(Exception exc)
   {
       System.out.println("Error! - " + exc.toString());
   }
}
```

Once again, the `StringTokenizer` comes to our rescue as we begin to decode the server's message to us in the `getAppointments` method. In order for the server to send us a message, we must prompt it to do so. That way, a socket connection is established, and a reply can be sent along the same route. It isn't entirely necessary to do things this way, but it is the preferred and time-honored method. Once we get our string from the server, we must tokenize it, step through each field, and convert it into a `Vector`.

```
public Vector getAppointments()
{
   // the variable to store all of our appointments in
   Vector appointmentVector = new Vector();
   // the string to put our appointments in
   String appointmentString = new String();
   // now get the appointments
   try
   {
      // tell the server we want the appointments it has
      outStream.println("retrieve|");
      // now listen for all the information we get back
      appointmentString = inStream.readLine();
   }
   catch(Exception exc)
   {
      System.out.println("Error! - " + exc.toString());
   }
   // tokenize the string
   StringTokenizer stk =
            new StringTokenizer(appointmentString, "|");
   // translate into a Vector
   while(stk.hasMoreTokens())
   {
```


```
        // create a variable to stick the appointment in
        AppointmentType appointment = new AppointmentType();
        // now get the next appointment from the string
        appointment.reason = stk.nextToken();
        appointment.time =
                 Integer.valueOf(stk.nextToken()).intValue();
        // put the appointment into the vector
        appointmentVector.addElement(appointment);
    }
    // return the Vector
    return appointmentVector;
}
```

Server

To implement the server, we will blatantly plagiarize code from the pizza application earlier in this chapter. Basically, we take all the server code from there, including the thread portion, and modify it for our needs. First, we need to implement the Store method. We will store our appointments in a `Vector` for simplicity's sake. The code snippet that follows is from the `Run` method of the `CalendarThread`.

NOTE: You could just as easily use some kind of serialization or even a file to keep your appointments persistent. When the server shuts down, we will lose all the appointments in our current implementation. Our server keeps data in a *transient* state, meaning that it is not maintained between executions.

```
// convert to a readable format
try
{
    StringTokenizer stk =
       new StringTokenizer(newOrder, "|");
    String operation = stk.nextToken();
    if(operation.equals("store"))
    {
        String reason = stk.nextToken();
        int time =
            Integer.valueOf(stk.nextToken()).intValue();
        // no exception was thrown so store the appointment
        AppointmentType appt = new AppointmentType();
        appt.reason = reason;
        appt.time = time;
        appointmentVector.addElement(appt);
        // put our result on the screen
        System.out.println("stored" + reason + "|" + time);
```

```
    }
}
catch(NoSuchElementException exc)
```

Now we must implement the retrieve function. The retrieve function creates a new string, delimited by the bar symbol, of course, that contains every appointment in our Vector. It then sends that information back to the client using the same socket on which it received the original message.

```
else
{
    String returnValue = new String();
    // put together a string of appointments
    for(int x = 0; x < appointmentVector.size(); x++)
    {
       AppointmentType appt  =
         (AppointmentType)appointmentVector.elementAt(x);
       returnValue += appt.reason + "|" + appt.time + "|";
    }
    // now write the appointments back to the socket
            out.println(returnValue);
}
```

Summary

Sockets are the backbone of any communication mechanism. Everything we talk about in this book from here on will use them in some way or another. For example, in the past some CORBA implementations used UDP for their socket infrastructure, eliminating complex webs of point-to-point connections. This sped up their implementation because they spent less time routing messages and more time sending them. When new objects were added to the system, UDP enabled them to be plugged in with little effort and little impact on the rest of the system. Lately, however, the onset of TCP-based IIOP has pushed almost all CORBA vendors to the more reliable protocol.

TCP is a reliable protocol system that has been used by generations of computer programmers. We all somehow, somewhere get our start in network programming by first using TCP/IP and writing to pipes and sockets. In the next chapter, we will explore Database access using Java Database Connectivity. JDBC is a technology that is basic to the concept of enterprise programming (i.e., tying our applications to our corporate databases). Although hidden from us by the API, at the very heart of JDBC are sockets. From JDBC we'll examine two examples of network object technologies. First we'll look at Java Remote Method Invocation, an all Java approach to distributed object computing. After RMI we'll look at the Java version of the grandfather of distributed object computing, CORBA. What we'll see when examining these technologies is that the abstractions of the object

models entirely hides the need to do socket level programming; this is done to simplify how we program. By eliminating the need to do our own socket programming, the abstractions provided by network object models provide a simpler programming model for us to deal with.