Section 11.4

Networking

As far as a program is concerned, a network is just another possible source of input data, and another place where data can be output. That does oversimplify things, because networks are not as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called java.net. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World-Wide Web, and provides the sort of network communication capability that is used by

a Web browser when it downloads pages for you to view. The main classes for this style of networking are java.net.URL and java.net.URLConnection. An object of type URL is an abstract representation of a Universal Resource Locator, which is an address for an HTML document or other resource on the Web. A URLConnection represents a network connection to such a resource.

The second style of I/O, which is more general and much more important, views the network at a lower level. It is based on the idea of a socket. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called java.net.Socket to represent sockets that are used for network communication. The term "socket" presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class Socket. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams.

11.4.1 URLs and URLConnections

The URL class is used to represent resources on the World-Wide Web. Every resource has an address, which identifies it uniquely and contains enough information for a Web browser to find the resource on the network and retrieve it. The address is called a "url" or "universal resource locator."

An object belonging to the URL class represents such an address. Once you have a URL object, you can use it to open a URLConnection to the resource at that address. A url is ordinarily specified as a string, such as "http://math.hws.edu/eck/index.html". There are also relative url's. A relative url specifies the location of a resource relative to the location of another url, which is called the base or context for the relative url. For example, if the context is given by the url http://math.hws.edu/eck/, then the incomplete, relative url "index.html" would really refer to http://math.hws.edu/eck/index.html.

An object of the class URL is not simply a string, but it can be constructed from a string representation of a url. A URL object can also be constructed from another URL object, representing a context, and a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName) throws
MalformedURLException
```

Note that these constructors will throw an exception of type MalformedURLException if the specified strings don't represent legal url's. The MalformedURLException class is a subclass of IOException, and it requires mandatory exception handling. That is, you must call the constructor inside a try..catch statement that handles the exception or in a subroutine that is declared to throw the exception.

The second constructor is especially convenient when writing applets. In an applet, two methods are available that provide useful URL contexts. The method getDocumentBase(), defined in the Applet and JApplet classes, returns an object of type URL. This URL represents the location from which the HTML page that contains the applet was downloaded. This allows the applet to go back and retrieve other files that are stored in the same location as that document. For example,

```
URL url = new URL(getDocumentBase(), "data.txt");
```

constructs a URL that refers to a file named data.txt on the same computer and in the same directory as the source file for the web page on which the applet is running. Another method, getCodeBase(), returns a URL that gives the location of the applet class file (which is not necessarily the same as the location of the document).

Once you have a valid URL object, you can call its openConnection() method to set up a connection. This method returns a URLConnection. The URLConnection object can, in turn, be used to create an InputStream for reading data from the resource represented by the URL. This is done by calling its getInputStream() method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The openConnection() and getInputStream() methods can both throw exceptions of type IOException. Once the InputStream has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as TextReader, or using a Scanner. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the URLConnection class is getContentType(), which returns a String that describes the type of information available from the URL. The return value can be null if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call getContentType() after getInputStream(). The string returned by getContentType() is in a format called a mime type. Mime types include "text/plain", "text/html", "image/jpeg", "image/gif", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "gif". If you are only interested in text data, for example, you can check whether the string returned by getContentType() starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and then copies the text onto the screen. Many of the operations in this subroutine can throw exceptions. They are handled by declaring that the subroutine "throws IOException" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws IOException {

   /* Open a connection to the URL, and get an input stream
      for reading data from the URL. */

   URL url = new URL(urlString);
   URLConnection connection = url.openConnection();
   InputStream urlData = connection.getInputStream();

   /* Check that the content is some type of text. */

   String contentType = connection.getContentType();
   if (contentType == null || contentType.startsWith("text") == false)
     throw new IOException("URL does not seem to refer to a text file.");

   /* Copy lines of text from the input stream to the screen, until
```

```
            end-of-file is encountered  (or an error occurs). */

         BufferedReader in;  // For reading from the connection's input stream.
         in = new BufferedReader( new InputStreamReader(urlData) );

         while (true) {
            String line = in.readLine();
            if (line == null)
               break;
            System.out.println(line);
         }

      } // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file ReadURL.java. When using the program, note that you have to specify a complete url, including the "http://" at the beginning. Here is an applet that does much the same thing. The applet lets you enter a URL, which can be either a complete URL or a relative URL. A relative URL will be interpreted relative to the document base of the applet. Error messages or text loaded from the URL will be displayed in the text area of the applet. (The amount of text is limited to 10000 characters.) When the applet starts up, it is configured to load the file ReadURL.java from this book's source code directory; just click the "Load" button:

You can also try to use this applet to look at the HTML source code for this very page. Just type s4.html into the input box at the bottom of the applet and then click on the Load button. You might want to experiment with other urls to see what types of errors can occur. For example, entering "bogus.html" is likely to generate a FileNotFoundException, since no document of that name exists in the directory that contains this page. As another example, you can probably generate a SecurityException by trying to connect to http://www.whitehouse.gov. (Not because it's an official secret -- any url that does not lead back to the same computer from which the applet was loaded will generate a security exception. To protect you from malicious applets, an applet is allowed to open network connections only back to the computer from which it came.) The source code for the applet is in the file ReadURLApplet.java.

11.4.2 TCP/IP and Client/Server

Communication over the Internet is based on a pair of protocols called the Transmission Control Protocol and the Internet Protocol, which are collectively referred to as TCP/IP. (In fact, there is a more basic communication protocol called UDP that can be used instead of TCP in certain applications. UDP is supported in Java, but for this discussion, I'll stick to the full TCP/IP, which provides reliable two-way communication between networked computers.)

For two programs to communicate using TCP/IP, each program must create a socket, as discussed earlier in this section, and those sockets must be connected. Once such a connection is made, communication takes place using input streams and output streams. Each program has its own input stream and its own output stream. Data written by one program to its output stream is transmitted to the other computer. There, it enters the input stream of the program at the other end of the network connection. When that program reads data from its input stream, it is receiving the data that was transmitted to it over the network.

The hard part, then, is making a network connection in the first place. Two sockets are involved. To get things started, one program must create a socket that will wait passively until a connection request comes in from another socket. The waiting socket is said to be listening for a connection. On the other side of the connection-to-be, another program creates a socket that sends out a connection request to the listening socket. When the listening socket receives the connection request, it responds, and the connection is established. Once that is done, each program can obtain an input stream and an output stream for sending data over the connection. Communication takes place through these streams until one program or the other closes the connection.

A program that creates a listening socket is sometimes said to be a server, and the socket is called a server socket. A program that connects to a server is called a client, and the socket that it uses to make a connection is called a client socket. The idea is that the server is out there somewhere on the network, waiting for a connection request from some client. The server can be thought of as offering some kind of service, and the client gets access to that service by connecting to the server. This is called the client/server model of network communication. In many actual applications, a server program can provide connections to several

clients at the same time. When a client connects to a server's listening socket, that socket does not stop listening. Instead, it continues listening for additional client connections at the same time that the first client is being serviced. To do this, it is necessary to use threads (Section 8.5). We'll look at how it works in the next section.

The URL class that was discussed at the beginning of this section uses a client socket behind the scenes to do any necessary network communication. On the other side of that connection is a server program that accepts a connection request from the URL object, reads a request from that object for some particular file on the server computer, and responds by transmitting the contents of that file over the network back to the URL object. After transmitting the data, the server closes the connection.

A client program has to have some way to specify which computer, among all those on the network, it wants to communicate with. Every computer on the Internet has an IP address which identifies it uniquely among all the computers on the net. Many computers can also be referred to by domain names such as math.hws.edu or www.whitehouse.gov. (See Section 1.7.) Traditional (or IPv4) IP addresses are 32-bit integers. They are usually written in the so-called "dotted decimal" form, such as 69.9.161.200, where each of the four numbers in the address represents an 8-bit integer in the range 0 through 255. A new version of the Internet Protocol, IPv6, is currently being introduced. IPv6 addresses are 128-bit integers and are usually written in hexadecimal form (with some colons and maybe some extra information thrown in). In actual use, IPv6 addresses are still fairly rare.

A computer can have several IP addresses, and can have both IPv4 and IPv6 addresses. Usually, one of these is the loopback address, which can be used when a program wants to communicate with another program *on the same computer*. The loopback address has IPv4 address 127.0.0.1 and can also, in general, be referred to using the domain name localhost. In addition, there can be one or more IP addresses associated with physical network connections. Your computer probably has some utility for displaying your computer's IP addresses. I have written a small Java program, ShowMyNetwork.java, that does the same thing. When I run ShowMyNetwork on my computer, the output is:

```
en1 :  /192.168.1.47  /fe80:0:0:0:211:24ff:fe9c:5271%5
lo0 :  /127.0.0.1  /fe80:0:0:0:0:0:0:1%1  /0:0:0:0:0:0:0:1%0
```

The first thing on each line is a network interface name, which is really meaningful only to the computer's operating system. The output also contains the IP addresses for that interface. In this example, lo0 refers to the loopback address, which has IPv4 address 127.0.0.1 as usual. The most important number here is 192.168.1.47, which is the IPv4 address that can be used for communication over the network.

Now, a single computer might have several programs doing network communication at the same time, or one program communicating with several other computers. To allow for this possibility, a network connection is actually identified by a port number in combination with an IP address. A port number is just a 16-bit integer. A server does not simply listen for connections -- it listens for connections *on a particular port*. A potential client must know both the Internet address (or domain name) of the computer on which the server is running and the port number on which the server is listening. A Web server, for example, generally

listens for connections on port 80; other standard Internet services also have standard port numbers. (The standard port numbers are all less than 1024, and are reserved for particular services. If you create your own server programs, you should use port numbers greater than 1024.)

11.4.3 Sockets

To implement TCP/IP connections, the java.net package provides two classes, ServerSocket and Socket. A ServerSocket represents a listening socket that waits for connection requests from clients. A Socket represents one endpoint of an actual network connection. A Socket can be a client socket that sends a connection request to a server. But a Socket can also be created by a server to handle a connection request from a client. This allows the server to create multiple sockets and handle multiple connections. A ServerSocket does not itself participate in connections; it just listens for connection requests and creates Sockets to handle the actual connections.

When you construct a ServerSocket object, you have to specify the port number on which the server will listen. The specification for the constructor is

```
public ServerSocket(int port) throws IOException
```

The port number must be in the range 0 through 65535, and should generally be greater than 1024. (A value of 0 tells the server socket to listen on any available port.) The constructor might throw a SecurityException

if a smaller port number is specified. An IOException can occur if, for example, the specified port number is already in use.

As soon as a ServerSocket is created, it starts listening for connection requests. The accept() method in the ServerSocket class accepts such a request, establishes a connection with the client, and returns a Socket that can be used for communication with the client. The accept() method has the form

```
public Socket accept() throws IOException
```

When you call the accept() method, it will not return until a connection request is received (or until some error occurs). The method is said to block while waiting for the connection. (While the method is blocked, the thread that called the method can't do anything else. However, other threads in the same program can proceed.) You can call accept() repeatedly to accept multiple connection requests. The ServerSocket will continue listening for connections until it is closed, using its close() method, or until some error occurs, or until the program is terminated in some way.

Suppose that you want a server to listen on port 1728, and suppose that you've written a method provideService(Socket) to handle the communication with one client. Then the basic form of the server program would be:

```
try {
   ServerSocket server = new ServerSocket(1728);
   while (true) {
```

```
            Socket connection = server.accept();
            provideService(connection);
        }
    }
    catch (IOException e) {
        System.out.println("Server shut down with error: " + e);
    }
```

On the client side, a client socket is created using a constructor in the Socket class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs.

Once you have a connected socket, no matter how it was created, you can use the Socket methods getInputStream() and getOutputStream() to obtain streams that can be used for communication over the connection. These methods return objects of type InputStream and OutputStream, respectively. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
/**
 * Open a client connection to a specified server computer and
```

```
 * port number on the server, and then do communication through
 * the connection.
 */
void doClientConnection(String computerName, int serverPort) {
   Socket connection;
   InputStream in;
   OutputStream out;
   try {
      connection = new Socket(computerName,serverPort);
      in = connection.getInputStream();
      out = connection.getOutputStream();
   }
   catch (IOException e) {
      System.out.println(
          "Attempt to create connection failed with error: " + e);
      return;
   }
    .
    .  // Use the streams, in and out, to communicate with server.
    .
   try {
```

```
            connection.close();
                // (Alternatively, you might depend on the server
                //  to close the connection.)
        }
        catch (IOException e) {
        }
    }  // end doClientConnection()
```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail here. However, what I've covered here should give you the basic ideas of network programming, and it is enough to write some simple network applications. Let's look at a few working examples of client/server programming.

11.4.4 A Trivial Client/Server

The first example consists of two programs. The source code files for the programs are DateClient.java and DateServer.java. One is a simple network client and the other is a matching server. The client makes a connection to the server, reads one line of text from the server, and displays that text on the screen. The text sent by the server consists of the current date and time on the computer where the server is running. In order to open a connection, the client must know the computer on which the server is running and the port on which it is listening. The server listens on port number 32007. The port number could be anything between 1025 and 65535, as long the server and the client use the same port. Port numbers between 1 and 1024 are reserved for standard services and should not be used for other servers. The name or IP number of the computer on which the server is running must be specified as a command-line argument For example, if the server is running on a computer named math.hws.edu, then you would typically run the client with the command "java DateClient math.hws.edu". Here is the complete client program:

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

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  The connection is made to
```

```
 * the port specified by LISTENING_PORT.  The program reads one
 * line of text from the connection and then closes the
 * connection.  It displays the text that it read on
 * standard output.  This program is meant to be used with
 * the server program, DateServer, which sends the current
 * date and time on the computer where the server is running.
 */
public class DateClient {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      String hostName;         // Name of the server computer to connect to.
      Socket connection;       // A socket for communicating with server.
       BufferedReader incoming; // For reading data from the connection.

       /* Get computer name from command line. */

       if (args.length > 0)
          hostName = args[0];
       else {
```

```
      // No computer name was given.  Print a message and exit.
   System.out.println("Usage:  java DateClient
                                  <server_host_name>");
   return;
}

/* Make the connection, then read and display a line of text. */

try {
   connection = new Socket( hostName, LISTENING_PORT );
   incoming = new BufferedReader(
                     new
        InputStreamReader(connection.getInputStream()) );
   String lineFromServer = incoming.readLine();
   if (lineFromServer == null) {
         // A null from incoming.readLine() indicates that
         // end-of-stream was encountered.
      throw new IOException("Connection was opened, " +
            "but server did not send any data.");
   }
   System.out.println();
```

```
            System.out.println(lineFromServer);
            System.out.println();
            incoming.close();
        }
        catch (Exception e) {
            System.out.println("Error:  " + e);
        }

    }  // end main()


} //end class DateClient
```

Note that all the communication with the server is done in a try..catch statement. This will catch the IOExceptions that can be generated when the connection is opened or closed and when data is read from the input stream. The connection's input stream is wrapped in a BufferedReader, which has a readLine() method that makes it easy to read one line of text. (See Subsection 11.1.4.)

In order for this program to run without error, the server program must be running on the computer to which the client tries to connect. By the way, it's possible to run the client and the server program on the same computer. For example, you can open two command windows, start the server in one window and then run the client in the other window. To make things like this easier, most computers will recognize the domain

name localhost and the IP number 127.0.0.1 as referring to "this computer." This means that the command "java DateClient localhost" will tell the DateClient program to connect to a server running on the same computer. If that command doesn't work, try "java DateClient 127.0.0.1".

The server program that corresponds to the DateClient client program is called DateServer. The DateServer program creates a ServerSocket to listen for connection requests on port 32007. After the listening socket is created, the server will enter an infinite loop in which it accepts and processes connections. This will continue until the program is killed in some way -- for example by typing a CONTROL-C in the command window where the server is running. When a connection is received from a client, the server calls a subroutine to handle the connection. In the subroutine, any Exception that occurs is caught, so that it will not crash the server. Just because a connection to one client has failed for some reason, it does not mean that the server should be shut down; the error might have been the fault of the client. The connection-handling subroutine creates a PrintWriter for sending data over the connection. It writes the current date and time to this stream and then closes the connection. (The standard class java.util.Date is used to obtain the current time. An object of type Date represents a particular date and time. The default constructor, "new Date()", creates an object that represents the time when the object is created.) The complete server program is as follows:

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

```
/**
 * This program is a server that takes connection requests on
 * the port specified by the constant LISTENING_PORT.  When a
 * connection is opened, the program sends the current time to
 * the connected socket.  The program will continue to receive
 * and process connections until it is killed (by a CONTROL-C,
 * for example).  Note that this server processes each connection
 * as it is received, rather than creating a separate thread
 * to process the connection.
 */
public class DateServer {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      ServerSocket listener;  // Listens for incoming connections.
     Socket connection;      // For communication with the connecting program.

     /* Accept and process connections forever, or until some error occurs.
        (Note that errors that occur while communicating with a connected
         program are caught and handled in the sendDate() routine, so
```

```
            they will not crash the server.) */

        try {
            listener = new ServerSocket(LISTENING_PORT);
            System.out.println("Listening on port " + LISTENING_PORT);
            while (true) {
                    // Accept next connection request and handle it.
               connection = listener.accept();
               sendDate(connection);
            }
        }
        catch (Exception e) {
            System.out.println("Sorry, the server has shut down.");
            System.out.println("Error:  " + e);
            return;
        }

    }  // end main()


    /**
```

```
    * The parameter, client, is a socket that is already connected to another
    * program.  Get an output stream for the connection, send the current time,
    * and close the connection.
    */
   private static void sendDate(Socket client) {
      try {
         System.out.println("Connection from " +
                                       client.getInetAddress().toString()
);
         Date now = new Date();  // The current date and time.
         PrintWriter outgoing;   // Stream for sending data.
         outgoing = new PrintWriter( client.getOutputStream() );
         outgoing.println( now.toString() );
         outgoing.flush();  // Make sure the data is actually sent!
         client.close();
      }
      catch (Exception e){
         System.out.println("Error: " + e);
      }
   } // end sendDate()
```

```
} //end class DateServer
```

When you run DateServer in a command-line interface, it will sit and wait for connection requests and report them as they are received. To make the DateServer service permanently available on a computer, the program really should be run as a daemon. A daemon is a program that runs continually on a computer, independently of any user. The computer can be configured to start the daemon automatically as soon as the computer boots up. It then runs in the background, even while the computer is being used for other purposes. For example, a computer that makes pages available on the World Wide Web runs a daemon that listens for requests for pages and responds by transmitting the pages. It's just a souped-up analog of the DateServer program! However, the question of how to set up a program as a daemon is not one I want to go into here. For testing purposes, it's easy enough to start the program by hand, and, in any case, my examples are not really robust enough or full-featured enough to be run as serious servers. (By the way, the word "daemon" is just an alternative spelling of "demon" and is usually pronounced the same way.)

Note that after calling out.println() to send a line of data to the client, the server program calls out.flush(). The flush() method is available in every output stream class. Calling it ensures that data that has been written to the stream is actually sent to its destination. You should generally call this function every time you use an output stream to send data over a network connection. If you don't do so, it's possible that the stream will collect data until it has a large batch of data to send. This is done for efficiency, but it can impose unacceptable delays when the client is waiting for the transmission. It is even possible that some of the data might remain untransmitted when the socket is closed, so it is especially important to call flush() before closing the connection. This is one of those unfortunate cases where different implementations of

Java can behave differently. If you fail to flush your output streams, it is possible that your network application will work on some types of computers but not on others.

11.4.5 A Simple Network Chat

In the DateServer example, the server transmits information and the client reads it. It's also possible to have two-way communication between client and server. As a first example, we'll look at a client and server that allow a user on each end of the connection to send messages to the other user. The program works in a command-line interface where the users type in their messages. In this example, the server waits for a connection from a single client and then closes down its listener so that no other clients can connect. After the client and server are connected, both ends of the connection work in much the same way. The user on the client end types a message, and it is transmitted to the server, which displays it to the user on that end. Then the user of the server types a message that is transmitted to the client. Then the client user types another message, and so on. This continues until one user or the other enters "quit" when prompted for a message. When that happens, the connection is closed and both programs terminate. The client program and the server program are very similar. The techniques for opening the connections differ, and the client is programmed to send the first message while the server is programmed to receive the first message. The client and server programs can be found in the files CLChatClient.java and CLChatServer.java. (The name "CLChat" stands for "command-line chat.") Here is the source code for the server:

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

/**
 * This program is one end of a simple command-line interface chat program.
 * It acts as a server which waits for a connection from the CLChatClient
 * program.  The port on which the server listens can be specified as a
 * command-line argument.  If it is not, then the port specified by the
 * constant DEFAULT_PORT is used.  Note that if a port number of zero is
 * specified, then the server will listen on any available port.
 * This program only supports one connection.  As soon as a connection is
 * opened, the listening socket is closed down.  The two ends of the connection
 * each send a HANDSHAKE string to the other, so that both ends can verify
 * that the program on the other end is of the right type.  Then the connected
 * programs alternate sending messages to each other.  The client always sends
 * the first message.  The user on either end can close the connection by
 * entering the string "quit" when prompted for a message.  Note that the first
 * character of any string sent over the connection must be 0 or 1; this
 * character is interpreted as a command.
 */
public class CLChatServer {

   /**
```

```
     * Port to listen on, if none is specified on the command line.
     */
    static final int DEFAULT_PORT = 1728;

    /**
     * Handshake string. Each end of the connection sends this  string to the
     * other just after the connection is opened.  This is done to confirm that
     * the program on the other side of the connection is a CLChat program.
     */
    static final String HANDSHAKE = "CLChat";

    /**
     * This character is prepended to every message that is sent.
     */
    static final char MESSAGE = '0';

    /**
     * This character is sent to the connected program when the user quits.
     */
    static final char CLOSE = '1';
```

```
public static void main(String[] args) {

   int port;   // The port on which the server listens.

   ServerSocket listener;  // Listens for a connection request.
   Socket connection;      // For communication with the client.

  BufferedReader incoming;  // Stream for receiving data from client.
   PrintWriter outgoing;     // Stream for sending data to client.
   String messageOut;        // A message to be sent to the client.
   String messageIn;         // A message received from the client.

   BufferedReader userInput; // A wrapper for System.in, for reading
                             // lines of input from the user.

   /* First, get the port number from the command line,
      or use the default port if none is specified. */

   if (args.length == 0)
      port = DEFAULT_PORT;
   else {
```

```
      try {
         port= Integer.parseInt(args[0]);
         if (port < 0 || port > 65535)
            throw new NumberFormatException();
      }
      catch (NumberFormatException e) {
         System.out.println("Illegal port number, " + args[0]);
         return;
      }
   }

   /* Wait for a connection request.  When it arrives, close
      down the listener.  Create streams for communication
      and exchange the handshake. */

   try {
      listener = new ServerSocket(port);
      System.out.println("Listening on port " +
                                    listener.getLocalPort());
      connection = listener.accept();
      listener.close();
```

```
            incoming = new BufferedReader(
                               new InputStreamReader
                                  (connection.getInputStream()) );
            outgoing = new PrintWriter(connection.getOutputStream());
            outgoing.println(HANDSHAKE);  // Send handshake to client.
            outgoing.flush();
          messageIn = incoming.readLine();  // Receive handshake from client.
            if (! HANDSHAKE.equals(messageIn) ) {
               throw new Exception("Connected program is not a CLChat!");
            }
          System.out.println("Connected.  Waiting for the first message.");
        }
        catch (Exception e) {
          System.out.println("An error occurred while opening connection.");
            System.out.println(e.toString());
            return;
        }

      /* Exchange messages with the other end of the connection until one side
         or the other closes the connection.  This server program waits for
         the first message from the client.  After that, messages alternate
            strictly back and forth. */
```

```
        try {
           userInput = new BufferedReader(new
                     InputStreamReader(System.in));
           System.out.println("NOTE: Enter 'quit' to end the program.\n");
           while (true) {
              System.out.println("WAITING...");
              messageIn = incoming.readLine();
              if (messageIn.length() > 0) {
                     // The first character of the message is a command.
If
                   // the command is CLOSE, then the connection is closed.
                     // Otherwise, remove the command character from the
                     // message and proceed.
                 if (messageIn.charAt(0) == CLOSE) {
                    System.out.println("Connection closed at other end.");
                    connection.close();
                    break;
                 }
                 messageIn = messageIn.substring(1);
              }
```

```
            System.out.println("RECEIVED:  " + messageIn);
            System.out.print("SEND:      ");
            messageOut = userInput.readLine();
            if (messageOut.equalsIgnoreCase("quit"))  {
                  // User wants to quit.  Inform the other side
                  // of the connection, then close the connection.
               outgoing.println(CLOSE);
               outgoing.flush();  // Make sure the data is sent!
               connection.close();
               System.out.println("Connection closed.");
               break;
            }
            outgoing.println(MESSAGE + messageOut);
            outgoing.flush(); // Make sure the data is sent!
            if (outgoing.checkError()) {
             throw new IOException("Error occurred while transmitting message.");
            }
         }
      }
      catch (Exception e) {
        System.out.println("Sorry, an error has occurred.  Connection lost.");
         System.out.println("Error:  " + e);
```

```
            System.exit(1);
        }

    }  // end main()


} //end class CLChatServer
```

This program is a little more robust than DateServer. For one thing, it uses a handshake to make sure that a client who is trying to connect is really a CLChatClient program. A handshake is simply information sent between client and server as part of setting up the connection, before any actual data is sent. In this case, each side of the connection sends a string to the other side to identify itself. The handshake is part of the protocol that I made up for communication between CLChatClient and CLChatServer. A protocol is a detailed specification of what data and messages can be exchanged over a connection, how they must be represented, and what order they can be sent in. When you design a client/server application, the design of the protocol is an important consideration. Another aspect of the CLChat protocol is that after the handshake, every line of text that is sent over the connection begins with a character that acts as a command. If the character is 0, the rest of the line is a message from one user to the other. If the character is 1, the line indicates that a user has entered the "quit" command, and the connection is to be shut down.

Remember that if you want to try out this program on a single computer, you can use two command-line windows. In one, give the command "java CLChatServer" to start the server. Then, in the other, use the command "java CLChatClient localhost" to connect to the server that is running on the same machine.

Section 11.5

Network Programming and Threads

In the previous section, we looked at several examples of network programming. Those examples showed how to create network connections and communicate through them, but they didn't deal with one of the fundamental characteristics of network programming, the fact that network communication is fundamentally asynchronous. From the point of view of a program on one end of a network connection, messages can arrive from the other side of the connection at any time; the arrival of a message is an *event* that is not under the control of the program that is receiving the message. Certainly, it is possible to design a network communication protocol that proceeds in a synchronous, step-by-step process from beginning to end -- but whenever the process gets to a point in the protocol where it needs to read a message from the other side of the connection, it has to *wait* for that message to arrive. Essentially, the process has to wait for a message-arrival event to occur before it can proceed. While it is waiting for the message, we say that the process is blocked.

Perhaps an event-oriented networking API would be a good approach to dealing with the asynchronous nature of network communication, but that is not the approach that is taken in Java (or, typically, in other languages). Instead, a serious network program in Java uses **threads**. Threads were introduced in Section 8.5. A thread is a separate computational process that can run in parallel with other threads. When a program uses threads to do network communication, it is possible that some threads will be blocked, waiting for incoming messages, but other threads will still be able to continue performing useful work.

11.5.1 A Threaded GUI Chat Program.

The command-line chat programs, CLChatClient.java and CLChatServer.java, from the previous section use a straight-through, step-by-step protocol for communication. After a user on one side of a connection enters a message, the user must wait for a reply from the other side of the connection. An asynchronous chat program would be much nicer. In such a program, a user could just keep typing lines and sending messages without waiting for any response. Messages that arrive -- asynchronously -- from the other side would be displayed as soon as they arrive. It's not easy to do this in a command-line interface, but it's a natural application for a graphical user interface. The basic idea for a GUI chat program is to create a thread whose job is to read messages that arrive from the other side of the connection. As soon as the message arrives, it is displayed to the user; then, the message-reading thread blocks until the next incoming message arrives. While it is blocked, however, other threads can continue to run. In particular, the GUI event-handling thread

that responds to user actions keeps running; that thread can send outgoing messages as soon as the user generates them.

In case this is not clear to you, here is an applet that simulates such a program. Enter a message in the input box at the bottom of the applet, and press return (or, equivalently, click the "Send" button):

Both incoming messages and messages that you send are posted to the JTextArea that occupies most of the applet. This is not a real network connection. When you send your first message, a separate thread is started by the applet. This thread **simulates** incoming messages from the other side of a network connection. In fact, it just chooses the messages at random from a pre-set list. At the same time, you can continue to enter and send messages. Of course, this applet doesn't really do any network communication, but the same idea can be used to write a GUI network chat program. The program GUIChat.java allows two-way network chatting that works similarly to this simulation, except that the incoming messages really do come from the other side of a network connection

The GUIChat program can act as both the client end and the server end of a connection. When GUIChat is started, a window appears on the screen. This window has a "Listen" button that the user can click to create a server socket that will listen for an incoming connection request; this makes the program act as a server. It also has a "Connect" button that the user can click to send a connection request; this makes the program act as a client. As usual, the server listens on a specified port number. The client needs to know the computer on

which the server is running and the port on which the server is listening. There are input boxes in the GUIChat window where the user can enter this information. Once a connection has been established between two GUIChat windows, each user can send messages to the other. The window has an input box where the user types the message. Pressing return while typing in this box sends the message. This means that the sending of the message is handled by the usual event-handling thread, in response to an event generated by a user action. Messages are received by a separate thread that just sits around waiting for incoming messages. This thread blocks while waiting for a message to arrive; when a message does arrive, it displays that message to the user. The window contains a large transcript area that displays both incoming and outgoing messages, along with other information about the network connection.

I urge you to compile the source code, GUIChat.java, and try the program. To make it easy to try it on a single computer, you can make a connection between one window and another window on the same computer, using "localhost" or "127.0.0.1" as the name of the computer. (Once you have one GUIChat window open, you can open a second one by clicking the "New" button.) I also urge you to read the source code. I will discuss only parts of it here.

The program uses a nested class, ConnectionHandler, to handle most network-related tasks. ConnectionHandler is a subclass of Thread. The ConnectionHandler thread is responsible for opening the network connection and then for reading incoming messages once the connection has been opened. (By putting the connection-opening code in a separate thread, we make sure that the GUI is not blocked while the connection is being opened. Like reading incoming messages, opening a connection is a blocking operation that can take some time to complete.) A ConnectionHandler is created when the user clicks the

"Listen" or "Connect" button. The "Listen" button should make the thread act as a server, while "Connect" should make it act as a client. To distinguish these two cases, the ConnectionHandler class has two constructors:

```
/**
 * Listen for a connection on a specified port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.  Note that the
 * thread will only listen for one connection, and then will
 * close its server socket.
 */
ConnectionHandler(int port) {
   state = ConnectionState.LISTENING;
   this.port = port;
   postMessage("\nLISTENING ON PORT " + port + "\n");
   start();
}

/**
 * Open a connection to specified computer and port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.
```

```
 */
ConnectionHandler(String remoteHost, int port) {
   state = ConnectionState.CONNECTING;
   this.remoteHost = remoteHost;
   this.port = port;
   postMessage("\nCONNECTING TO " + remoteHost + " ON PORT " + port +
"\n");
   start();
}
```

Here, state is an instance variable whose type is defined by an enumerated type

```
enum ConnectionState { LISTENING, CONNECTING, CONNECTED, CLOSED };
```

The values of this enum represent different possible states of the network connection. It is often useful to treat a network connection as a state machine (see Subsection 6.5.4), since the response to various events can depend on the state of the connection when the event occurs. Setting the state variable to LISTENING or CONNECTING tells the thread whether it should act as a server or as a client. Note that the postMessage() method posts a message to the transcript area of the window, where it will be visible to the user.

Once the thread has been started, it executes the following run() method:

```
/**
 * The run() method that is executed by the thread.  It opens a
 * connection as a client or as a server (depending on which
 * constructor was used).
 */
public void run() {
   try {
      if (state == ConnectionState.LISTENING) {
            // Open a connection as a server.
         listener = new ServerSocket(port);
         socket = listener.accept();
         listener.close();
      }
      else if (state == ConnectionState.CONNECTING) {
            // Open a connection as a client.
         socket = new Socket(remoteHost,port);
      }
      connectionOpened();  // Sets up to use the connection (including
                           // creating a BufferedReader, in, for reading
                           // incoming messages).
      while (state == ConnectionState.CONNECTED) {
```

```
            // Read one line of text from the other side of
            // the connection, and report it to the user.
         String input = in.readLine();
         if (input == null)
            connectionClosedFromOtherSide();
         else
            received(input);  // Report message to user.
      }
   }
   catch (Exception e) {
         // An error occurred.  Report it to the user, but not
         // if the connection has been closed (since the error
         // might be the expected error that is generated when
         // a socket is closed).
      if (state != ConnectionState.CLOSED)
         postMessage("\n\n ERROR:  " + e);
   }
   finally {  // Clean up before terminating the thread.
      cleanUp();
   }
}
```

This method calls several other methods to do some of its work, but you can see the general outline of how it works. After opening the connection as either a server or client, the run() method enters a while loop in which it receives and processes messages from the other side of the connection until the connection is closed. It is important to understand how the connection can be closed. The GUIChat window has a "Disconnect" button that the user can click to close the connection. The program responds to this event by closing the socket that represents the connection. It is likely that when this happens, the connection-handling thread is blocked in the in.readLine() method, waiting for an incoming message. When the socket is closed by another thread, this method will fail and will throw an exception; this exception causes the thread to terminate. (If the connection-handling thread happens to be between calls to in.readLine() when the socket is closed, the while loop will terminate because the connection state changes from CONNECTED to CLOSED.) Note that closing the window will also close the connection in the same way.

It is also possible for the user on the other side of the connection to close the connection. When that happens, the stream of incoming messages ends, and the in.readLine() on this side of the connection returns the value null, which indicates end-of-stream and acts as a signal that the connection has been closed by the remote user.

For a final look into the GUIChat code, consider the methods that send and receive messages. These methods are called from different threads. The send() method is called by the event-handling thread in response to a user action. Its purpose is to transmit a message to the remote user. It uses a PrintWriter, out, that writes to the socket's output stream. Synchronization of this method prevents the connection state from changing in the middle of the send operation:

```
/**
 * Send a message to the other side of the connection, and post the
 * message to the transcript.  This should only be called when the
 * connection state is ConnectionState.CONNECTED; if it is called at
 * other times, it is ignored.
 */
synchronized void send(String message) {
   if (state == ConnectionState.CONNECTED) {
      postMessage("SEND:  " + message);
      out.println(message);
      out.flush();
      if (out.checkError()) {
         postMessage("\nERROR OCCURRED WHILE TRYING TO SEND DATA.");
         close();  // Closes the connection.
      }
   }
}
```

The received() method is called by the connection-handling thread **after** a message has been read from the remote user. Its only job is to display the message to the user, but again it is synchronized to avoid the race condition that could occur if the connection state were changed by another thread while this method is being executed:

```
/**
 * This is called by the run() method when a message is received from
 * the other side of the connection.  The message is posted to the
 * transcript, but only if the connection state is CONNECTED.  (This
 * is because a message might be received after the user has clicked
 * the "Disconnect" button; that message should not be seen by the
 * user.)
 */
synchronized private void received(String message) {
   if (state == ConnectionState.CONNECTED)
      postMessage("RECEIVE:  " + message);
}
```

11.5.2 A Multithreaded Server

There is still one big problem with the GUIChat program. In order to open a connection to another computer, a user must know that there is a GUIChat program listening on some particular port on some particular computer. Except in rather contrived situations, there is no way for a user to know that. It would be nice if it were possible to discover, somehow, who's out there on the Internet waiting for a connection. Unfortunately, this is not possible. And yet, applications such as AOL Instant Messenger seem to do just that -- they can show you a list of users who are available to receive messages. How can they do that?

I don't know the details of instant messenger protocols, but it has to work something like this: When you start the client program, that program contacts a server program that runs constantly on some particular computer and on some particular port. Since the server is always available at the same computer and port, the information needed to contact it can be built into the client program or otherwise made available to the users of the program. The purpose of the server is to keep a list of available users. When your client program contacts the server, it gets a list of available users, along with whatever information is necessary to send messages to those users. At the same time, your client program registers you with the server, so that the server can tell other users that you are on-line. When you shut down the client program, you are removed from the server's list of users, and other users can be informed that you have gone off-line.

Of course, in an application like AOL server, you only get to see a list of available users from your "buddy list," a list of your friends who are also AOL users. To implement this, you need to have an account on the AOL server. The server needs to keep a database of information about all user accounts, including the buddy list for each user. This makes the server program rather complicated, and I won't consider that aspect of its functionality here. However, it is not very difficult to write a scaled-down application that uses the network in a similar way. I call my scaled-down version "BuddyChat." It doesn't keep separate buddy lists for each user; it assumes that you're willing to be buddies with anyone who happens to connect to the server. In this application, the server keeps a list of connected users and makes that list available to each connected user. A user can connect to another user and chat with that user, using a window that is very similar to the chat window in GUIChat. BuddyChat is still just a toy, compared to serious network applications, but it does illustrate some core ideas.

The BuddyChat application comes in several pieces. BuddyChatServer.java is the server program, which keeps the list of available users and makes that list available to clients. Ideally, the server program would run constantly (as a daemon) on a computer and port that are known to all the possible client users. For testing, of course, it can simply be stated like any other program. The client program is BuddyChat.java. This program is to be run by any user who wants to use the BuddyChat service. When a user starts the client program, it connects to the server, and it gets from the server a list of other users who are currently connected. The list is displayed to the user of the client program, who can send a request for a chat connection with any user on the list. The client can also receive incoming chat connection requests from other users. The window that is used for chatting is defined by BuddyChatWindow.java, which is not itself a program but just a subclass of JFrame that defines the chat window. (There is also a fourth piece, BuddyChatServerShutdown.java. This is a program that can be run to shut down the BuddyChatServer gracefully. I will not discuss it further here. See the source code for more information, if you are interested.)

I urge you to compile the programs and try them out. For testing, you can try them on a single computer (although all the windows can get a little confusing). First, start BuddyChatServer. The server has no GUI interface, but it does print some information to standard output as it runs. Then start the BuddyChat client program. When BuddyChat starts up, it presents a window where you can enter the name and port number for the server and your "handle," which is just a name that will identify you in the server's list of users. The server info is already set up to connect to a server on the same machine. When you hit the "Connect" button, a new window will open with a list, currently empty, of other users connected to the server. Now, start another copy of the BuddyChat client program. When you click "Connect", you'll have two client list

windows, one for each copy of the client program that you've started. (One of these windows will be exactly on top of the other, so you'll have to move it to see the second window.) Each client window will display the other client in its list of users. You can run additional copies of the client program, if you want, and you might want to try connecting from another computer if one is available.

At this point, there is a network connection in place between the server and each client. Whenever a client connects to or disconnects from the server, the server sends a notification of the event to each connected client, so that the client can modify its own list of connected users. The server also maintains a listening socket that listens for connection requests from new clients. In order to manage all this, the server is running several threads. One thread waits for connection requests on the listening socket. In addition to this, there are two threads for each connected client -- one thread for sending messages to the client and one thread for reading messages sent by the client to the server.

Back to trying out the program. Remember that the whole point was to provide each user with a list of potential chat partners. Click on a user in one of the client user lists, and then click the "Connect to Selected Buddy" button. When you do this, your BuddyChat program sends a connection request to the BuddyChat program that is being run by the selected user. Each BuddyChat program, one on each side of the connection, opens a chat window (of type BuddyChatWindow). A network connection between these two windows is set up without any further action on the part of the two users, and the users can use the windows to send messages back and forth to each other. The BuddyChatServer program has nothing to do with opening, closing, or using the connection between its two clients (although a different design might have had the messages go through the server).

In order to open the chat connection from one program to another, the second program must be listening for connection requests and the first program must know the computer and port on which the first user is listening. In the BuddyChat system, the BuddyChatServer knows this information and provides it to each BuddyChat client program. The **users** of the client programs never have to be aware of this information.

How does the server know about the clients' computers and port numbers? When a BuddyChat client program is run, in addition to opening a connection to the BuddyChatServer, the client also creates a listening socket to accept connection requests from other users. When the client registers with the server, it tells the server the port number of the client's listening socket. The server also knows the IP address of the computer on which the client is running, since it has a network connection to that computer. This means that the BuddyChatServer knows the IP address and listening socket port number of every BuddyChat client. A copy of this information is provided (along with the users' handles) to each connected client program. The net result is that every BuddyChat client program has the information that it needs to contact all the other clients.

The basic techniques used in the BuddyChat system are the same as those used in previous networking examples: server sockets, client sockets, input and output streams for sending messages over the network, and threads to handle the communication. The important difference is how these basic building blocks are combined to build a more complex application. I have tried to explain the logic of that application here. I will not discuss the BuddyChat source code here, since it is locally similar to examples that we have already looked at, but I encourage you to study the source code if you are interested in network programming.

BuddyChat seems to have a lot of functionality, yet I said it was still a "toy" program. What exactly makes it a toy? There are at least two big problems. First of all, it is not scalable. A network program is scalable if it will work well for a large number of simultaneous users. BuddyChat would have problems with a large number of users because it uses so many threads (two for each user). It takes a certain amount of processing for a computer to switch its attention from one thread to another. On a very busy server, the constant switching between threads would soon start to degrade the performance. One solution to this is to use a more advanced network API. Java has a class SelectableChannel that makes it possible for one thread to manage communication over a large number of network connections. This class is part of the package java.nio that provides a number of advanced I/O capabilities for working with files and networking. However, I will not cover those capabilities in this book.

But the biggest problem is that BuddyChat offers absolutely no defense against denial of service attacks. In a denial of service, a malicious user attacks a network server in some way that prevents other users from accessing the service or severely degrades the performance of the service for those users. It would be simple to launch a denial of service attack on BuddyChat by making a huge number of connections to the server. The server would then spend most of its time servicing those bogus connections. The server could guard against this to some extent by putting a limit on the number of simultaneous connections that it will accept from a given IP address. It would also be helpful to add some security to the server by requiring users to know a password in order to connect. However, neither of these measures would fully solve the problem, and it is very difficult to find a complete defense against denial of service attacks.

11.5.3 Distributed Computing

In Section 8.5, we saw how threads can be used to do parallel processing, where a number of processors work together to complete some task. In that section, it was assumed that all the processors were inside one multi-processor computer. But parallel processing can also be done using processors that are in different computers, as long as those computers are connected to a network over which they can communicate. This type of parallel processing -- in which a number of computers work together on a task and communicate over a network -- is called distributed computing.

In some sense, the whole Internet is an immense distributed computation, but here I am interested in how computers on a network can cooperate to solve some computational problem. There are several approaches to distributed computing that are supported in Java. RMI and CORBA are standards that enable a program running on one computer to call methods in objects that exist on other computers. This makes it possible to design an object-oriented program in which different parts of the program are executed on different computers. RMI (Remote Method Invocation) only supports communication between Java objects. CORBA (Common Object Request Broker Architecture) is a more general standard that allows objects written in various programming languages, including Java, to communicate with each other. As is commonly the case in networking, there is the problem of locating services (where in this case, a "service" means an object that is available to be called over the network). That is, how can one computer know which computer a service is located on and what port it is listening on? RMI and CORBA solve this problem using something like our little BuddyChatServer example -- a server running at a known location keeps a list of services that are

available on other computers. Computers that offer services register those services with the server; computers that need services contact the server to find out where they are located.

RMI and CORBA are complex systems that are not very easy to use. I mention them here because they are part of Java's standard network API, but I will not discuss them further. Instead, we will look at a relatively simple demonstration of distributed computing that uses only basic networking.

The problem that we will look at uses the simplest type of parallel programming, in which the problem can be broken down into tasks that can be performed independently, with no communication between the tasks. To apply distributed computing to this type of problem, we can use one "master" program that divides the problem into tasks and sends those tasks over the network to "worker" programs that do the actual work. The worker programs send their results back to the master program, which combines the results from all the tasks into a solution of the overall problem. In this context, the worker programs are often called "slaves," and the program uses the so-called master/slave approach to distributed computing.

The demonstration program is defined by three source code files: CLMandelbrotMaster.java defines the master program; CLMandelbrotWorker.java defines the worker programs; and CLMandelbrotTask.java defines the class, CLMandelbrotTask, that represents an individual task that is performed by the workers. To run the demonstration, you must start the CLMandelbrotWorker program on several computers (probably by running it on the command line). This program uses CLMandelbrotTask, so both class files, CLMandelbrotWorker.class and CLMandelbrotTask.class, must be present on the worker computers. You can then run CLMandelbrotMaster on the master computer. Note that this program also requires the class

CLMandelbrotTask. You must specify the host name or IP address of each of the worker computers as command line arguments for CLMandelbrotMaster. A worker program listens for connection requests from the master program, and the master program must be told where to send those requests. For example, if the worker program is running on three computers with IP addresses 172.30.217.101, 172.30.217.102, and 172.30.217.103, then you can run CLMandelbrotMaster with the command

```
java CLMandelbrotMaster 172.30.217.101 172.30.217.102 172.30.217.103
```

The master will make a network connection to the worker at each IP address; these connections will be used for communication between the master program and the workers.

It is possible to run several copies of CLMandelbrotWorker on the same computer, but they must listen for network connections on different ports. It is also possible to run CLMandelbrotWorker on the same computer as CLMandelbrotMaster. You might even see some speed-up when you do this, if your computer has several processors. See the comments in the program source code files for more information, but here are some commands that you can use to run the master program and two copies of the worker program on the same computer. Give these commands in separate command windows:

```
java CLMandelbrotWorker                    (Listens on default
port)
```

```
java  CLMandelbrotWorker  1501                          (Listens on port
1501)

java  CLMandelbrotMaster  localhost  localhost:1501
```

Every time CLMandelbrotMaster is run, it solves exactly the same problem. (For this demonstration, the nature of the problem is not important, but the problem is to compute the data needed for a picture of a small piece of the famous "Mandelbrot Set." If you are interested in seeing the picture that is produced, uncomment the call to the saveImage() method at the end of the main() routine in CLMandelbrotMaster.java. We will encounter the Mandelbrot Set again as an example in Chapter 12.)

You can run CLMandelbrotMaster with different numbers of worker programs to see how the time required to solve the problem depends on the number of workers. (Note that the worker programs continue to run after the master program exists, so you can run the master program several times without having to restart the workers.) In addition, if you run CLMandelbrotMaster with no command line arguments, it will solve the entire problem on its own, so you can see how long it takes to do so without using distributed computing. In a trial that I ran, it took 40 seconds for CLMandelbrotMaster to solve the problem on its own. Using just one worker, it took 43 seconds. The extra time represents extra work involved in using the network; it takes time to set up a network connection and to send messages over the network. Using two workers (on different computers), the problem was solved in 22 seconds. In this case, each worker did about half of the work, and their computations were performed in parallel, so that the job was done in about half the time. With larger numbers of workers, the time continued to decrease, but only up to a point. The master program itself has a

certain amount of work to do, no matter how many workers there are, and the total time to solve the problem can never be less than the time it takes for the master program to do its part. In this case, the minimum time seemed to be about five seconds.

Let's take a look at how this distributed application is programmed. The master program divides the overall problem into a set of tasks. Each task is represented by an object of type CLMandelbrotTask. These tasks have to be communicated to the worker programs, and the worker programs must send back their results. Some protocol is needed for this communication. I decided to use character streams. The master encodes a task as a line of text, which is sent to a worker. The worker decodes the text (into an object of type CLMandelbrotTask) to find out what task it is supposed to perform. It performs the assigned task. It encodes the results as another line of text, which it sends back to the master program. Finally, the master decodes the results and combines them with the results from other tasks. After all the tasks have been completed and their results have been combined, the problem has been solved.

The problem is divided into a fairly large number of tasks. A worker receives not just one task, but a sequence of tasks. Each time it finishes a task and sends back the result, it is assigned a new task. After all tasks are complete, the worker receives a "close" command that tells it to close the connection. In CLMandelbrotWorker.java, all this is done in a method named handleConnection() that is called to handle a connection that has already been opened to the master program. It uses a method readTask() to decode a task

that it receives from the master and a method writeResults() to encode the results of the task for transmission back to the master. It must also handle any errors that occur:

```
private static void handleConnection(Socket connection) {
   try {
      BufferedReader in = new BufferedReader( new InputStreamReader(
                                          connection.getInputStream())
);
      PrintWriter out = new PrintWriter(connection.getOutputStream());
      while (true) {
         String line = in.readLine();  // Message from the master.
         if (line == null) {
               // End-of-stream encountered -- should not happen.
            throw new Exception("Connection closed unexpectedly.");
         }
         if (line.startsWith(CLOSE_CONNECTION_COMMAND)) {
               // Represents the normal termination of the connection.
            System.out.println("Received close command.");
            break;
         }
         else if (line.startsWith(TASK_COMMAND)) {
               // Represents a CLMandelbrotTask that this worker is
```

```
                // supposed to perform.
             CLMandelbrotTask task = readTask(line);  // Decode the message.
              task.compute();  // Peform the task.
              out.println(writeResults(task));  //  Send back the results.
              out.flush();
           }
           else {
                // No other messages are part of the protocol.
              throw new Exception("Illegal command received.");
           }
        }
     }
     catch (Exception e) {
        System.out.println("Client connection closed with error " + e);
     }
     finally {
        try {
           connection.close();  // Make sure the socket is closed.
        }
        catch (Exception e) {
        }
```

```
    }
}
```

Note that this method is **not** executed in a separate thread. The worker has only one thing to do at a time and does not need to be multithreaded.

You might wonder why so many tasks are used. Why not just divide the problem into one task for each worker? The reason is that using a larger number of tasks makes it possible to do load balancing. Not all tasks take the same amount of time to execute. This is true for many reasons. Some of the tasks might simply be more computationally complex than others. Some of the worker computers might be slower than others. Or some worker computers might be busy running other programs, so that they can only give part of their processing power to the worker program. If we assigned one task per worker, it is possible that a complex task running on a slow, busy computer would take much longer than the other tasks to complete. This would leave the other workers idle and delay the completion of the job while that worker completes its task. To complete the job as quickly as possible, we want to keep all the workers busy and have them all finish at about the same time. This is called load balancing. If we have a large number of tasks, the load will automatically be approximately balanced: A worker is not assigned a new task until it finishes the task that it is working on. A slow worker, or one that happens to receive more complex tasks, will complete fewer tasks than other workers, but all workers will be kept busy until close to the end of the job. On the other hand, individual tasks shouldn't be too small. Network communication takes some time. If it takes longer to transmit a task and its results than it does to perform the task, then using distributed computing will take

more time than simply doing the whole job on one computer! A problem is a good candidate for distributed computing if it can be divided into a fairly large number of fairly large tasks.

Turning to the master program, CLMandelbrotMaster.java, we encounter a more complex situation. The master program must communicate with several workers over several network connections. To accomplish this, the master program is multi-threaded, with one thread to manage communication with each worker. A pseudocode outline of the main() routine is quite simple:

```
create a list of all tasks that must be performed
if there are no command line arguments {
      // The master program does all the tasks itself.
   Perform each task.
}
else {
      // The tasks will be performed by worker programs.
   for each command line argument:
      Get information about a worker from command line argument.
      Create and start a thread to communicate with the worker.
   Wait for all threads to terminate.
}
// All tasks are now complete (assuming no error occurred).
```

The list of tasks is stored in a variable, tasks, of type ArrayList<CLMandelbrotTask>. The communication threads take tasks from this list and send them to worker programs. The method getNextTask() gets one task from the list. If the list is empty, it returns null as a signal that all tasks have been assigned and the communication thread can terminate. Since tasks is a resource that is shared by several threads, access to it must be controlled; this is accomplished by writing getNextTask() as a synchronized method:

```
synchronized private static CLMandelbrotTask getNextTask() {
   if (tasks.size() == 0)
      return null;
   else
      return tasks.remove(0);
}
```

(The reason for the synchronization is to avoid the race condition that could occur between the time that the value of tasks.size() is tested and the time that tasks.remove() is called. See Subsection 8.5.3 for information about parallel programming, race conditions, and synchronized.)

The job of a thread is to send a sequence of tasks to a worker thread and to receive the results that the worker sends back. The thread is also responsible for opening the connection in the first place. A pseudocode outline for the process executed by the thread might look like:

```
Create a socket connected to the worker program.
```

```
Create input and output streams for communicating with the worker.
while (true) {
   Let task = getNextTask().
   If task == null
      break;  // All tasks have been assigned.
   Encode the task into a message and transmit it to the worker.
   Read the response from the worker.
   Decode and process the response.
}
Send a "close" command to the worker.
Close the socket.
```

This would work OK. However, there are a few subtle points. First of all, the thread must be ready to deal with a network error. For example, a worker might shut down unexpectedly. But if that happens, the master program can continue, provided other workers are still available. (You can try this when you run the program: Stop one of the worker programs, with CONTROL-C, and observe that the master program still completes successfully.) A difficulty arises if an error occurs while the thread is working on a task: If the problem as a whole is going to be completed, that task will have to be reassigned to another worker. I take care of this by putting the uncompleted task back into the task list. (Unfortunately, my program does not handle all possible errors. If a network connection "hangs" indefinitely without actually generating an error,

my program will also hang, waiting for a response from a worker that will never arrive. A more robust program would have some way of detecting the problem and reassigning the task.)

Another defect in the procedure outlined above is that it leaves the worker program idle while the thread is processing the worker's response. It would be nice to get a new task to the worker before processing the response from the previous task. This would keep the worker busy and allow two operations to proceed simultaneously instead of sequentially. (In this example, the time it takes to process a response is so short that keeping the worker waiting while it is done probably makes no significant difference. But as a general principle, it's desirable to have as much parallelism as possible in the algorithm.) We can modify the procedure to take this into account:

```
try {
   Create a socket connected to the worker program.
   Create input and output streams for communicating with the worker.
   Let currentTask = getNextTask().
   Encode currentTask into a message and send it to the worker.
   while (true) {
      Read the response from the worker.
      Let nextTask = getNextTask().
      If nextTask != null {
            // Send nextTask to the worker before processing the
            // response to currentTask.
```

```
            Encode nextTask into a message and send it to the worker.
         }
         Decode and process the response to currentTask.
         currentTask = nextTask.
         if (currentTask == null)
            break; // All tasks have been assigned.
      }
      Send a "close" command to the worker.
      Close the socket.
   }
   catch (Exception e) {
      Put uncompleted task, if any, back into the task list.
   }
   finally {
      Close the connection.
   }
```

Finally, here is how this translates into Java. The pseudocode presented above becomes the run() method in the class that defines the communication threads used by the master program:

```
/**
 * This class represents one worker thread.  The job of a worker thread
```

```
 * is to send out tasks to a CLMandelbrotWorker program over a network
 * connection, and to get back the results computed by that program.
 */
private static class WorkerConnection extends Thread {

   int id;         // Identifies this thread in output statements.
   String host;    // The host to which this thread will connect.
   int port;       // The port number to which this thread will connect.

   /**
    * The constructor just sets the values of the instance
    * variables id, host, and port and starts the thread.
    */
   WorkerConnection(int id, String host, int port) {
      this.id = id;
      this.host = host;
      this.port = port;
      start();
   }

   /**
```

```
     * The run() method of the thread opens a connection to the host and
     * port specified in the constructor, then sends tasks to the
     * CLMandelbrotWorker program on the other side of that connection.
     * If the thread terminates normally, it outputs  the number of tasks
     * that it processed.  If it terminates with an error, it outputs
     * an error message.
     */
    public void run() {

       int tasksCompleted = 0; // How many tasks has this thread handled.
       Socket socket;  // The socket for the connection.

       try {
          socket = new Socket(host,port);  // open the connection.
       }
       catch (Exception e) {
         System.out.println("Thread " + id + " could not open connection to " +
                 host + ":" + port);
          System.out.println("   Error: " + e);
          return;
       }
```

```
CLMandelbrotTask currentTask = null;
CLMandelbrotTask nextTask = null;

try {
   PrintWriter out = new PrintWriter(socket.getOutputStream());
   BufferedReader in = new BufferedReader(
                          new
                     InputStreamReader(socket.getInputStream()) );
   currentTask = getNextTask();
   if (currentTask != null) {
         // Send first task to the worker program.
      String taskString = writeTask(currentTask);
      out.println(taskString);
      out.flush();
   }
   while (currentTask != null) {
     String resultString = in.readLine(); // Get results for currentTask.
      if (resultString == null)
         throw new IOException("Connection closed unexpectedly.");
      if (! resultString.startsWith(RESULT_COMMAND))
       throw new IOException("Illegal string received from worker.");
```

```
      nextTask = getNextTask();  // Get next task and send it to worker.
        if (nextTask != null) {
            // Send nextTask to worker before processing results for
            // currentTask, so that the worker can work on nextTask
              // while the currentTask results are processed.
           String taskString = writeTask(nextTask);
           out.println(taskString);
           out.flush();
        }
        readResults(resultString, currentTask);
      finishTask(currentTask);  // Process results from currentTask.
        tasksCompleted++;
      currentTask = nextTask;   // We are finished with old currentTask.
        nextTask = null;
    }
   out.println(CLOSE_CONNECTION_COMMAND);  // Send close command to worker.
    out.flush();
}
catch (Exception e) {
   System.out.println("Thread " + id + " terminated because of an error");
    System.out.println("   Error: " + e);
    e.printStackTrace();
```

```
            // Put uncompleted task, if any, back into the task list.
         if (currentTask != null)
            reassignTask(currentTask);
         if (nextTask != null)
            reassignTask(nextTask);
      }
      finally {
        System.out.println("Thread " + id + " ending after completing " +
               tasksCompleted + " tasks");
         try {
            socket.close();
         }
         catch (Exception e) {
         }
      }

   } //end run()

} // end nested class WorkerConnection
```