[Previous Section | Next Section | Chapter Index | Main Index]

Section 11.5

Network Programming and Threads

Subsections

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
```

[Previous Section | Next Section | Chapter Index | Main Index]