Module 16

Networking

Objectives

At the end of this module, you should be able to:

- Develop code to set up the network connection
- Understand the TCP/IP Protocol
- Use `ServerSocket` and `Socket` classes for implementation of TCP/IP clients and servers

This module discusses Java 2 SDK support for sockets and socket programming. Socket programming communicates with other programs running on computers on the same network.

Relevance

Discussion – The following question is relevant to the material presented in this module:

How can a communication link between a client machine and a server on the network be established?

Networking

The following section describes the concept of networking by using sockets.

Sockets

Socket is the name given, in one particular programming model, to the endpoints of a communication link between processes. Because of the popularity of that particular programming model, the term socket has been reused in other programming models, including Java technology.

When processes communicate over a network, Java technology uses the streams model. A socket can hold two streams: one input stream and one output stream. A process sends data to another process through the network by writing to the output stream associated with the socket. A process reads data written by another process by reading from the input stream associated with the socket.

After the network connection is set up, using the streams associated with that connection is similar to using any other stream.

Setting Up the Connection

To set up the connection, one machine must run a program that is waiting for a connection, and a second machine must try to reach the first. This is similar to a telephone system, in which one party must make the call, while the other party is waiting by the telephone when that call is made.

A description of the TCP/IP network connections is presented in this module. An example network connection is shown in Figure 16-1.

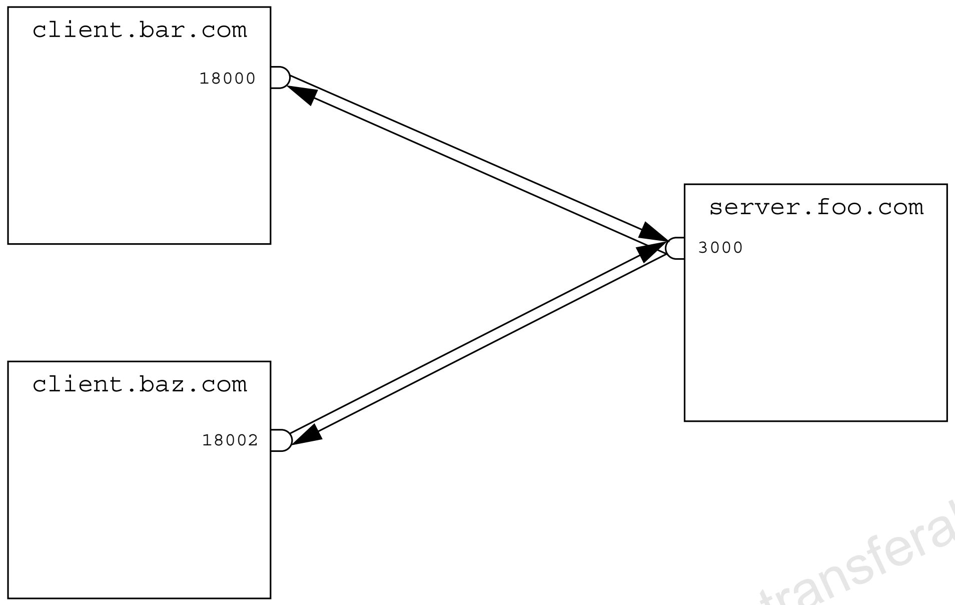

Figure 16-1 Diagram of Example Network Connections

Networking With Java Technology

This section describes the concept of networking by using Java technology.

Addressing the Connection

When you make a telephone call, you need to know the telephone number to dial. When you make a network connection, you need to know the address or the name of the remote machine. In addition, a network connection requires a port number, which you can think of as a telephone extension number. After you connect to the proper computer, you must identify a particular purpose for the connection. So, in the same way that you can use a particular telephone extension number to talk to the accounts department, you can use a particular port number to communicate with the accounting program.

Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and the values range from 0–65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless communicating with one of those services (such as `telnet`, Simple Mail Transport Protocol [SMTP] mail, `ftp`, and so on). Client port numbers are allocated by the host OS to something not in use, while server port numbers are specified by the programmer, and are used to identify a particular service.

Both client and server must agree in advance on which port to use. If the port numbers used by the two parts of the system do not agree, communication does not occur.

Java Networking Model

In the Java programming language, TCP/IP socket connections are implemented with classes in the `java.net` package. Figure 16-2 illustrates what occurs on the server side and the client side.

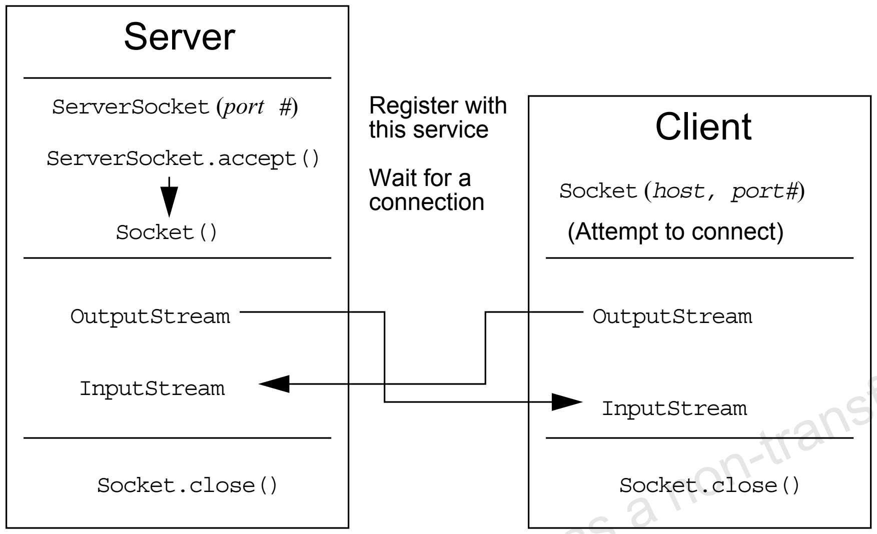

Figure 16-2 TCP/IP Socket Connections

In Figure 16-2:

- The server assigns a port number. When the client requests a connection, the server opens the socket connection with the `accept()` method.
- The client establishes a connection with *`host`* on port *`port#`*.
- Both the client and server communicate by using an `InputStream` and an `OutputStream`.

Minimal TCP/IP Server

TCP/IP server applications rely on the `ServerSocket` and `Socket` networking classes provided by the Java programming language. The `ServerSocket` class takes most of the work out of establishing a server connection.

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

public class SimpleServer {
  public static void main(String args[]) {
    ServerSocket s = null;

    // Register your service on port 5432
    try {
      s = new ServerSocket(5432);
    } catch (IOException e) {
      e.printStackTrace();
    }

   // Run the listen/accept loop forever
    while (true) {
      try {
        // Wait here and listen for a connection
        Socket s1 = s.accept();

        // Get output stream associated with the socket
        OutputStream s1out = s1.getOutputStream();
        BufferedWriter bw = new BufferedWriter(
          new OutputStreamWriter(s1out));

        // Send your string!
        bw.write("Hello Net World!\n");

        // Close the connection, but not the server
socket
        bw.close();
        s1.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Minimal TCP/IP Client

The client side of a TCP/IP application relies on the Socket class. Again, much of the work involved in establishing connections is done by the Socket class. The client attaches to the server presented in "Minimal TCP/IP Server" on page 16-7, and then prints everything sent by the server to the console.

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

public class SimpleClient {
  public static void main(String args[]) {
    try {
      // Open your connection to a server, at port 5432
      // localhost used here
      Socket s1 = new Socket("127.0.0.1", 5432);

      // Get an input stream from the socket
      InputStream is = s1.getInputStream();
      // Decorate it with a "data" input stream
      DataInputStream dis = new DataInputStream(is);

      // Read the input and print it to the screen
      System.out.println(dis.readUTF());

      // When done, just close the steam and connection
      br.close();
      s1.close();
    } catch (ConnectException connExc) {
      System.err.println("Could not connect.");
    } catch (IOException e) {
      // ignore
    }
  }
}
```