

---

## 29. Network Programming

*Have more than thou showest,  
Speak less than thou knowest.  
-William Shakespeare*

Network programming is a term of the field that refers to communication-based programming. Typically, we write a program that makes use of an API that facilitates communication between one or more processes. Often, the processes can run on different physical CPUs.

As mentioned in Chapter 15, when there are multiple CPU's involved with a computation, we have a concurrent programming environment. We can classify the type of network programming that we are doing based on its location in the OSI layers.

In this chapter you will learn how the TCP/IP suite can be used to perform network computing. You will learn:

- The OSI Reference Model
- How to use domain name servers from Java
- How to write a web server
- How to write a browser
- How to use Java to set the time from an atomic clock
- How to use Java to get a stock quote
- How to send instances of classes across the Internet for execution
- How to send mail using Java

### 29.1 The OSI Reference Model

Figure 29.1-1 shows a sketch of the OSI reference model.

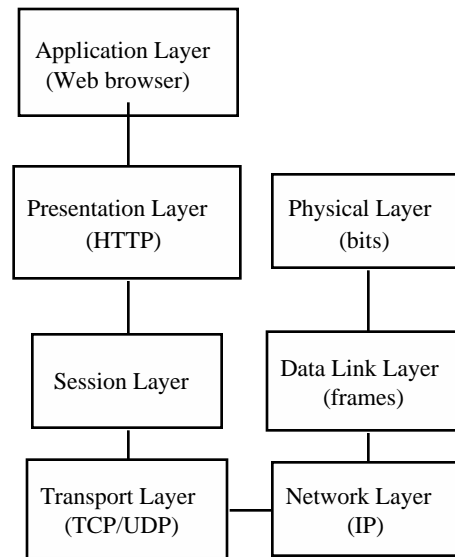


Figure 29.1-1. The OSI Reference Model

In the physical layer, the basic physical layer data unit is the bit. Issues of how the bits are transmitted (via various energy modulation techniques) are beyond the scope of this book.

In the data link layer, the basic data link layer data unit is a *frame*. The frame is surrounded by header information (start and stop bits). It also may contain error correcting or error detecting codes (e.g., parity, or cyclic redundancy check bits). Data link layer communications transmit data from one point to another. For example, PPP (Point to Point Protocol) is used by modems.

In the network layer (what some people call, the *Internet* layer), IP (Internet Protocol) addresses are used to communicate. The communication can involve routing between different subnetworks (hence the term *internetworking*).

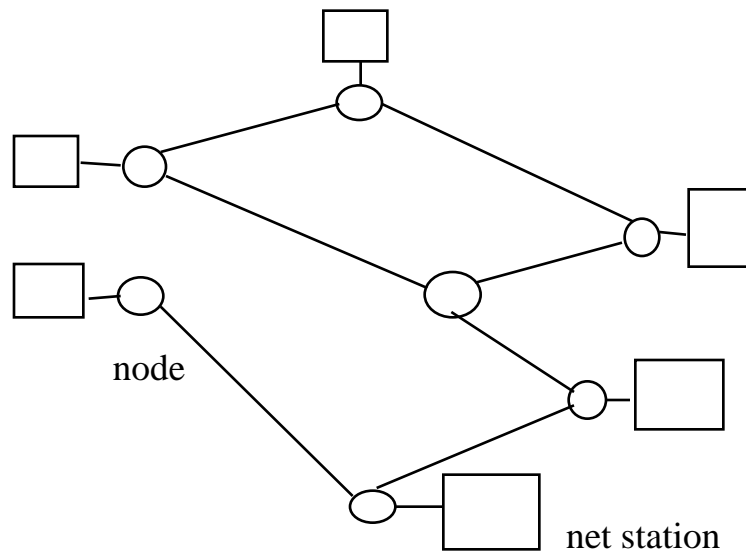


Figure 29.1-2. An Internetworking System

Figure 29.1-2 shows a sketch of an internetworking system. The square blocks represent network stations. The circles represent nodes on the network. The nodes are able to route internet protocol data units (e.g., packets) to the correct next node. This distributed routing feature is reserved for the *network* layer. Placing a single IP addressed packet into a shared media is called a packet data switched network. Packet switched nets are store and forward nets. A packet is stored and forwarded at each node along its path. In comparison, a circuit switched network has a dedicated communication path. For example, a telephone can have a dedicated path between one hand set and the next.

In the transport layer the services of flow control are performed. While the network layer routes packets from one node to the next, the TCP (Transmission Control Protocol) will make sure that the packets arrive in the correct order. Thus, we can treat the TCP/IP services as a bit-pipe, with bits going in one and bits coming out at the other, in the correct order.

The session layer is used to open and close connections. The protocol for a telephone, for example, is to pick up the hand set, wait for the dial tone, dial, wait for the ringing and the answer, then begin the session. The protocol may involve a login, with a password.

The presentation layer permits data representation consistency. For example, in the *ftp* (File Transfer Protocol), carriage returns are mapped to line feeds or line feeds are mapped to carriage returns, depending on the platform defaults. Sometimes a presentation layer protocol is used to transport a specific kind of data. For example, hypertext transfer protocol (HTTP) is used to transport HTML.

The application layer provides a user interface. For example, a browser, kermit, ftp, Telnet, etc.

## 29.2 The Client-Server Paradigm

The client-server paradigm lies at the root of the modern *distributed computing environment*. In such a milieu, processing functions reside in different address spaces (clients or servers). A front-end runs on a client, a back-end runs on a server. A distributed computing environment uses network protocols and spreads its logic across the server and client. Such a programming technique has the advantages of improved flexibility, scalability, and extendibility. It can lower software development cost.

Scalability is accomplished via adding more servers to improve performance. This leads to load balancing and fault tolerance. Fault tolerance leads to improved reliability.

## 29.3 The Dns Class

A *domain name* is an hierarchical namespace that comes in the form of period (“.”) delimited identifiers. For example *www.docjava.com* is a domain name. The top-level domain is *com*.

Domain Name	Meaning
com	commercial organizations
edu	educational institutions
gov	government institutions
mil	military groups
net	network support centers
org	organizations other than the above
biz	Businesses
info	Unrestricted use
name	For registration by individuals
pro	Accountants, lawyers, and physicians
museum	Museums
country code	geographic country scheme

Figure 29.3-1. Top-level domains.

Figure 29.3-1 shows a listing of top-level domains. The *biz*, *info*, *name*, *pro* and *museum* are new top-level domains. To learn more about top-level domains, see <http://www.icann.org/tlds/>. To view the 239 country codes, see [http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en\\_listp1.html](http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html). To get a country code, you must be on the United Nations list of countrys. This number has grown from 51, in 1945 to 189 in year 2000 <http://www.un.org>.

All official domain names end with a top-level domain. Inside an *intranet* (i.e., a network not directly tied to the Internet) it is possible to create unofficial domain names. The domain name owner typically has control over the parts of the domain name that precede the top-level domain. For example, DocJava, Inc. owns the *docjava.com* domain and controls the *www.docjava.com* mapping. This section describes how to perform the mapping between domain names and addresses using Java. An Internet address is a period (“.”) delimited set of four numbers that range from 1 to 254 called *dotted decimal notation*. For example: 192.168.1.1 is a valid address. A domain name server performs the service of mapping the domain name to an Internet address. Domain name servers use protocols that enable them to communicate their results. The *DNS* class provides an API that uses these protocols to communicate with domain name servers.

Mapping a machine name into an IP address is the first step toward getting a connection to a machine. This is often done implicitly by many of the classes.

```
package net;
import java.net.*;

public class Dns {
```

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

The *Dns* class has a single static method called *printAddress*. It serves as an example of how to map the machine name into an IP address.

```
    Dns.printAddress("www.docjava.com");
}
public static void printAddress(String name) {
try {
```

The *InetAddress* class resides in the *java.net* package. The address that is returned is often of the form 192.168.1.100 (that is a dotted list of 4 number that ranges from 1 to 254).

```
    InetAddress ia =
        InetAddress.getBy_name(name);
byte IP[] = ia.getAddress();
for (int index = 0; index < IP.length; index++) {
    if (index > 0) System.out.print(".");
```

The *0xff* mask makes sure that the numbers do not print as negative numbers when they are converted to *int*:

```
        System.out.print(((int)IP[index])&0xff);
    }
    System.out.println();
}
catch (UnknownHostException e) {
    System.out.println("Feh, that was not a valid name!");
}
}
}
```

## 29.4 The WebServer Class and Sockets

This section shows how to create a web server for testing with a browser. The web server uses a *ServerSocket* instance. A *ServerSocket* provides a communication end-point that is used to listen for a connection request. During the listening process, the thread of execution is blocked. The thread becomes unblocked when a request comes in. Thus, the *ServerSocket* is IO bound and must be used inside of a thread.

When communicating using sockets, you need both an IP (Internet Protocol) address and a *port* number. The IP address is typically assigned to the server and mapped to a machine name. For example, the name *www.nsf.gov* maps to the IP address 128.150.4.107. At that address there are open ports, willing to

accept connections. One of those ports is port 80. Port 80 is a well known port, used, by default, for web servers.

To understand which services are available at which ports, type:

```
more /etc/services
```

at a Unix prompt. On my Redhat 7.1 system, this file has over 500 lines of service descriptions. For example:

```
...
echo          7/tcp
echo          7/udp
discard       9/tcp          sink null
discard       9/udp          sink null
sysstat       11/tcp         users
sysstat       11/udp         users
daytime       13/tcp
daytime       13/udp
...
```

This shows that *ping* is on port 7, and that the day and time services are on port 13. If a port is already in use, it is a contention error to try to listen on it. Thus, it is the server programmer's job to find a free port for their service.

When a request for a connection is obtained by a server, the server creates a *socket*. The socket arranges the data in the transport layer of the OSI reference model. The data streams in and out, using a connection oriented protocol. The job of arranging the data-packets that arrive out of order into sequential order requires a buffer and latency. Also, an algorithm (called *Nagle's algorithm*) is used to improve throughput at a cost of increased latency. The latency is controllable using the *Socket.setTcpNoDelay(true)* method.

The following abbreviated listing chops off the *import* statements. To gain access to the classes you will have to import *java.net.\**. Also, the classes of this chapter are in the *net* package:

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

The *main* method allows the starting of the web server at any port. Port 80 is the default port

```
        WebServer ws = new WebServer(80);
    }
```

The server socket blocks the main thread of execution, until a connection request is entered.

```
WebServer(int port) {
    try {
        ServerSocket ss
            = new ServerSocket(port);
```

As soon as a connection request is obtained, we issued the *ServerSocket.accept* method. This returns an instance of a *Socket* class. The *Socket* instance enables us to have access to input streams and output streams.

```
        while (true) {
            startClient(ss.accept());
        }
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

For every connection, we start a thread. Thus, this is a multi-threaded web server.

```
public void startClient(Socket s)
    throws IOException {
    ClientThread
        ct = new ClientThread(s);
    Thread t = new Thread(ct);
    t.start();
}
}
```

The *WebServer* class forms a framework that is generally correct, no matter what the protocol is (i.e., we could write any type of server using this technique). The *ClientThread* implements *Runnable* because it is used as an argument to the *Thread* constructor. The *Socket* instance is used for the source of both the *BufferedReader* and the *PrintWriter*. That means that *Sockets* provide two-way communications (called *full-duplex*):

```
class ClientThread implements
    Runnable {
    Socket s;
    BufferedReader br;
    PrintWriter pw;
```

The *http* protocol is defined in <http://www.faqs.org/rfcs/rfc2068.html>. The protocol consists of several strings. These are treated as constants in the program.

```
public static final String
    notFoundString =
```

```

"HTTP/1.0 501 Not Implemented\n"
+"Content-type: text/plain\n\n";
public static final String
okString =
"HTTP/1.0 200 OK\n"
+"Content-type: text/html\n\n";

```

Given an instance of a *Socket* the following code shows how to obtain an *InputStream* instance and an *OutputStream* instance. These are going to transport text information, and so a *BufferedReader* and *PrintWriter* are used for input and output.

```

ClientThread(Socket _s) {
    s = _s;
    try{
        br =
            new BufferedReader(
                new InputStreamReader(
                    s.getInputStream()));
        pw =
            new PrintWriter(
                s.getOutputStream(), true);
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}

```

When reading a line of text from the *BufferedReader* instance, the server expects to see a series of client commands. There are a series of predefined *Request Methods* that are permitted as commands in the *http*. Our simple server only supports the *GET* request. If the *GET* request is made, we answer by counting to 10 (a very simple response!).

```

public void run() {
    try {
        String line =
            br.readLine();
        StringTokenizer
            st =
                new StringTokenizer(
                    line);
        System.out.println(line);
        if (st.nextToken().equals("GET"))
            //getAFile(st);
            countTo10();
        else
            pw.println(notFoundString);
        pw.close();
        s.close();
    }
    catch(Exception e) {

```

```
    }
}
```

The *countTo10* method answers by supplying the *http* header and a *MIME-TYPE*. MIME stands for Multipurpose Internet Mail Extensions. MIME is used to permit the transport of non-text message bodies via e-mail. It has been extended to map to a number of different applications. The supplied *MIME-TYPE* must be understood by the browser. Every browser is different. Often browsers have helper applications or plug-ins that assist them in interpreting data of a specific MIME Type. See

<<http://trade.chonbuk.ac.kr/~leesl/rfc/rfc1521.html>> for more information about MIME types.

```
public void countTo10() {
    pw.println("HTTP/1.0 200 OK");
    pw.println("Content-type: text/plain");
    pw.println();
    for (int i=0; i < 10; i++)
        pw.println("i="+i);
}
```

The *getAFile* method is a first crack at supplying a file to the client, without first checking what type the file is. This is a basic *text* file service. In order to first determine the file type, we might like to use the *StreamSniffer* given in Chapter 19.

```
public void getAFile(StringTokenizer st)
    throws FileNotFoundException,
        IOException {
    pw.println(okString);
    String fileName
        = st.nextToken();
    pw.println("date="+new java.util.Date());
    BufferedReader
        fileReader= new
            BufferedReader(new
                FileReader(
                    "d:\\www\\"+fileName));
    String fileLine=null;
    while(
        (fileLine =
            fileReader.readLine())
        != null)
        pw.println(fileLine);
    }
}
```

To understand the operation of the web server better, use the *telnet* command to telnet into port 80 of the web server. Type *GET* and hit enter twice. You should get a response. You can also visit the web-server with a browser.

## 29.5 The Browser Class and URLs

In this section we show how to build your very own web browser. This is important because it provides an illustration of a new class called the *URL* class. It also shows how you can use the Web as a source of data. The output of the program, after it visits the `<http://www.docjava.com>` web page is:

```
web browser started
url=http://www.docjava.com/
Vector print
<!--This file created 9/26/00 10:05 AM by Claris Home Page
      version 2.0-->
<HTML>
<HEAD>
  <TITLE>Distance Learning</TITLE>
```

.....*text deleted for brevity*....

```
</BODY>
</HTML>
```

The code for the *Browser* class follows:

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

The *url* is hard coded into the program for demonstration purposes. This is a simple web browser, and so it does not attempt to provide an interface.

```
public class Browser {
    public static void main(String args[]) {
        System.out.println("web browser started");

        print(
            getUrl("http://www.docjava.com"));
    }
}
```

The *Browser.getUrl* method is a generally useful one that will return a *Vector* of all the lines read from a *URL*. The *getUrl* method will block the current thread of execution, so it might be wise to put it in a thread, if the connection is slow.

```
public static Vector getUrl(String
    _urlString) {
    try {
```

```

        URL url
            = new URL(_urlString);
        System.out.println("url="
            +url);
        return
            getUrl(url);
    }
    catch(Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

The *getUrl* method runs in a *for* loop until the last line is read.

```

public static Vector getUrl(
    URL url)
    throws IOException {
    Vector v = new Vector();
    BufferedReader br
        = new BufferedReader(
            new InputStreamReader(
                url.openStream()));
    for (String l=br.readLine();
        l != null;
        l=br.readLine())
        v.addElement(l);
    return v;
}
public static void print(Vector v) {
    System.out.println("Vector print");
    for (int i=0;i < v.size(); i++)
        System.out.println(
            (String)v.elementAt(i));
}
}

```

Adding a graphic user interface to the browser is a simple matter, particularly because we already have the *HTMLViewer* in the *gui* package.

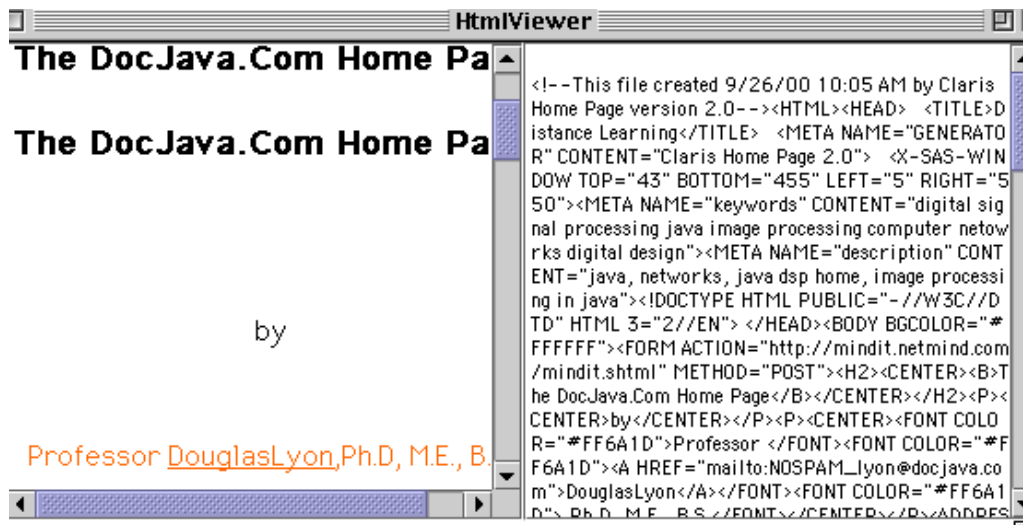


Figure 29.5-1. The HtmlViewer

Figure 29.5-1 shows an image of the *HtmlViewer* along with the rendered HTML. A small change to the *Browser* class enables the display:

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

    gui.HtmlViewer hv = new gui.HtmlViewer();
    String s =
        Browser.toString("http://www.docjava.com");
    hv.setHtml(s);
    hv.setText(s);
}
```

The new *toString* class takes the *url* string and concatenates a large string, with all the HTML in it. This is returned and rendered in the *HtmlViewer*.

```
public static String toString(String url) {
    Vector v = getUrl(url);
    String s = "\n";
    for (int i=0; i < v.size(); i++)
        s = s + v.elementAt(i);
    return s;
}
```

## 29.6 The Quote Class

Based on our work with the *Browser* class, it is a simple feature to get a stock quote from the Internet. The hard part of the program is to know where to go to get the stock quotes. Try typing the following URL:

```
<http://quote.yahoo.com/download/javasoft.beans?SYMBOLS=aapl&
format=sl>
```

The output is:

```
"AAPL",22.70,"7/9/2001","4:00PM",+0.67,22.09,23,21.68,6026200
```

To formulate these URL's to communicate with the *yahoo* service, we use the

*Quote* class:

```
package net;
import java.util.Vector;
import java.util.StringTokenizer;
import java.util.Date;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.net.URL;
import java.awt.*;
import java.awt.event.*;

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

First, we need a list of valid symbols of trading stocks. These symbols can be NYSE, AMEX or NASDAQ:

```
        Vector s = new Vector();
        s.addElement("aapl");
        s.addElement("xlnx");
        s.addElement("altr");
        s.addElement("mot");
        s.addElement("cy");
        s.addElement("crus");
        s.addElement("sfa");
        s.addElement("adbe");
        s.addElement("msft");
        s.addElement("sunw");
        Browser.print(
            Browser.getUrl(makeQuoteURLString(s)));
    }
```

The heart of the program is the *makeQuoteURLString*, which iterates through each symbol, creating comma separated elements (i.e.,

“SYMBOLS=aapl,ibm...”):

```
public static String makeQuoteURLString(Vector symbols) {
    String symbolsString = "";
    for(int i = 0; i < symbols.size(); i++) {
        String symbol = (String)symbols.elementAt(i);
        symbolsString += ((i != 0) ? "," : "")
            + symbol.toUpperCase();
    }
    return

        "http://quote.yahoo.com/download/javasoft.beans?SYMBOL
        S="
```

```

    + symbolsString
    + "&format=sl";
}
}

```

The output of the program follows:

```

url=http://quote.yahoo.com/download/javasoft.beans?SYMBOLS=AA
    PL,XLNX,ALTR,MOT,CY,CRUS,SFA,ADBE,MSFT,SUNW&format=sl
Vector print
"AAPL",22.70,"7/9/2001","4:00PM",+0.67,22.09,23,21.68,6026200
"XLNX",37.45,"7/9/2001","4:00PM",-
    0.21,37.80,38.43,36.65,4993600
"ALTR",27.28,"7/9/2001","4:00PM",+0.67,26.65,27.71,26.40,4731
    500
"MOT",15.37,"7/9/2001","4:00PM",0.00,15.21,15.79,15.04,959150
    0
"CY",21.34,"7/9/2001","4:01PM",-
    0.44,21.08,21.93,21.08,1390400
"CRUS",25.14,"7/9/2001","4:00PM",+0.89,24.35,25.34,24.16,1168
    000
"SFA",42.36,"7/9/2001","4:01PM",+0.76,42.28,43.30,42.02,24569
    00
"ADBE",43.98,"7/9/2001","4:00PM",+0.62,43.02,44.20,42.88,4681
    300
"MSFT",65.69,"7/9/2001","4:00PM",-
    0.37,66.20,66.91,65.04,33238500
"SUNW",14.82,"7/9/2001","4:00PM",+1.14,14.18,14.95,13.97,4334
    4900

```

## 29.7 The AtomicClock Class

The *AtomicClock* class uses an atomic clock to get, and set, the time. Port 13 is a well known port for getting the day and time. As an experiment, try typing:

```
telnet time-A.timefreq.bldrdoc.gov 13
```

Under *Unix* you get:

```
Trying 132.163.4.101...
```

```
Connected to time-A.timefreq.bldrdoc.gov.
```

```
Escape character is '^]'.
```

After a few carriage returns, the time is printed:

```
52100 01-07-10 09:52:54 50 0 0 515.2 UTC(NIST) *
```

Then the connection is closed. In this section we open a socket to the atomic clock server, parse the response and then set our system clock to the time.

To set up the program to set the systems time, place the following lines in a file called *d:\Java\GetTime\SetSysTime.bat*:

```
REM This batch file sets the system time to the value input
    on the command line

time=%1
```

We will invoke this from Java (the setting part of the code only works under *Windows*, but the code runs and executes on any platform).

```
package net;
/**
 * Class:      GetTimeConsole
 * @author:    James Linn, D. Lyon
 *
 * Date:      1/20/2001
 * Time:      11:14:12 PM
 */

import java.text.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class AtomicClock {
```

The time server will take an *unknown* amount of time to respond, via the Internet. As a result, we will not be able to know the time with great precision:

```
public static BufferedReader getReader()
    throws UnknownHostException, IOException {
    Socket s=new Socket("time-
A.timefreq.bldrdoc.gov",13);
    return new BufferedReader(
        new InputStreamReader(s.getInputStream()));
}
```

The following code sets the time using the *SetSysTime.bat* file described above. If you are a non-windows machine, this code will not work. However, you can still invoke it!

```
public static void setSystemTime(GregorianCalendar c)
    throws IOException {
    // Now set the computer to this time:
    Runtime rt = Runtime.getRuntime();
    StringTokenizer st = new StringTokenizer(
c.getTime().toString(), " ");
    for(int i=0;i<3;i++)
        st.nextToken();

    String[] execArgArray=new String[2];
    execArgArray[0]="d:\\Java\\GetTime\\SetSysTime.bat";
    execArgArray[1]=st.nextToken();
    rt.exec(execArgArray);
}
```

The following method takes the return from the time server and parses it into an instance of a *GregorianCalendar* class.

```

public static GregorianCalendar parseDate(String line)
{
    StringTokenizer sTok=new StringTokenizer(line," ");
    sTok.nextToken(); // Get # at the beginning,
    whatever it represents
    String sDate=sTok.nextToken();
    String sTime=sTok.nextToken();
    SimpleDateFormat df=new SimpleDateFormat("yy-MM-dd
    hh:mm:ss");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    GregorianCalendar cal=new GregorianCalendar();
    cal.setTimeZone(TimeZone.getTimeZone("GMT"));
    try {
        cal.setTime(df.parse(sDate+" "+sTime));
    }
    catch(ParseException e) {
        System.out.println(e.getMessage());
    }
    System.out.println(cal.getTime());
    return cal;
}

```

If the *batch file* method of setting the system time is unsuitable for your computer, this is the place to change it. The *setSystemTime* invocation can be replaced with something compatible for your system. Java has no way to set the system clock in its core API. A native method could be used for this application. Also, on some systems (i.e., Unix, NT Server, etc.) the setting of the system clock is privileged.

```

public static void processLine(String s)
    throws IOException {
    if (s.length() == 0) return;
    System.out.println(s);
    setSystemTime(parseDate(s));
    sleep();
}

```

This is the primary loop of the program. We keep reading lines until there are no more lines to read. *Sleep* is needed or else we tend to get time-outs in the connection.

```

public static void getTime()
    throws UnknownHostException, IOException {
    BufferedReader in = getReader();
    String s = null;
    while((s=in.readLine())!=null)
        processLine(s);
}
public static void sleep() {
    try {
        Thread.sleep(5000);
    }
}

```

```
        // Gives time to read msg
    }
    catch(InterruptedException ie) {
    }
}
public static void main(String[] args) {
    try {
        AtomicClock.getTime();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
}
```

The program output follows:

```
52100 01-07-10 10:52:48 50 0 0 354.1 UTC(NIST) *
Tue Jul 10 06:52:48 EDT 2001
```

## 29.8 Object Oriented Services

In this section we show how to use sockets to pass instances of classes across the network. The primary advantage is that a strongly typed object can be passed. This eliminates the need for parsing (like we saw in Section 29.7). It also provides for remote invocation of methods, thus enabling true concurrency.

Another feature of the code in this section is that the server and client are communicating via sockets, but both reside in the same virtual machine. This has the effect of simplifying set-up and test of concurrent programming systems. It is typically much easier to debug a client-server computing system if you take a single-computer multi-threaded approach for debugging. For example:

```
package net;

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

Our *MainServer* class serves as a mediator that launches both the *serverThread* and the date client. Because the system runs so quickly, a *race-condition* occurs if we start the server and then the client right away. That is because it takes time to set up the *ServerSocket*. A *race-condition* is a transiently inconsistent state that occurs because we did not incorporate a notification call back to notify us when the *ServerSocket* is in place. As a result, we incorporate a 1000 ms (1 second) wait after we launch the sever thread:

```

public class MainServer implements Runnable {
    public static void main(String args[]) {
        Thread serverThread = new Thread(new MainServer());
        serverThread.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}

```

Now that the server is awake, we can test the server:

```

DateClient dc = new DateClient();
dc.run();
dc.run();
dc.run();
}

```

The *MainServer* starts a new *DateServer* thread every time it gets a connection. In this way, it should be unlikely for any *dateClient* to be denied access, since there is always a *ServerSocket* instance accepting connections:

```

public void run() {
    try {
        ServerSocket ss =
            new ServerSocket(13);
        while (true) {
            System.out.println("waiting");
            Socket socket = ss.accept();
            DateServer ds =
                new DateServer(socket);
            ds.start();
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

The *DateServer* runs in its own thread, serving the requests of the *DateClient*:

```

package net;

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

```

The *DateServer* has an instance of the *ObjectOutputStream* class. This is used to write instances of reference types that implement *Serializable*:

```

public class DateServer extends Thread {
    ObjectOutputStream oos;

    public DateServer(Socket s)
        throws IOException {

```

```

    oos =
        new ObjectOutputStream(
            s.getOutputStream());
}

```

By writing an instance of the *Date* to the *ObjectOutputStream* instance, we pass a full reference data type across the socket. This is better than a remote procedure call, because the instance and its implementation are passed:

```

public void run() {
    try {
        oos.writeObject(new Date());
        oos.close();
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}
}

```

Now we present the *DateClient*. The communication is based on our fore-knowledge about the type of instance that we expect from the server. The host name *localhost* and the port *13* may need to be changed if you want the program to work on two different machines. Port 13 is typically a day-time port used by *Unix*.

```

package net;

import java.util.Date;
import java.net.*;
import java.io.*;
public class DateClient implements Runnable {
    public void run() {
        try {
            Socket s
                = new Socket("localhost",13);

```

When an instance is read from an *ObjectInputStream* instance, it comes in as a reference of type *Object*. We use *instanceof* to make sure that the instance is a *Date*, before we cast it.

```

    ObjectInputStream
        ois = new ObjectInputStream(
            s.getInputStream());
    Object o =ois.readObject();
    if (o instanceof Date)
        System.out.println((Date)o);
    ois.close();
}
catch(Exception e) {
    e.printStackTrace();
}
}

```

```
}
```

## 29.9 The Compute Server

This section shows how to perform full-duplex (i.e., two-way) communication between the client and the server so that computations can be performed. Ideally, we should only have to set up our computation server once. After configuration, it becomes a network resource that permits anyone who has access to our network to access the compute server(s). The steps involved in the distributed computation are as follows:

On the server:

1. Read an instance of the *ComputableObject*
2. Perform the computation
3. Send back the answer.

On the client:

1. Send an instance of the *ComputableObject*
2. Get back an instance of the answer.
3. Process the answer.

The computation server must be multi-threaded (just like a web server) so that it will not block during computations. The client must be able to stop and wait for an answer so that functions like:

```
a = f(x);
```

will execute atomically. In order to minimize the configuration problems, we propose an interface that every *ComputableObject* implements:

```
package net;

import java.io.*;

public interface ComputableObject extends Serializable {
    public Object compute();
}
```

An example implementation follows:

```
package net;

public class ComputeMe implements ComputableObject {
    public Object compute() {
        return new Integer(4*4+3);
    }
}
```

```
    }
}
```

In order to stress test our system, we would like to create several computation clients. In order to accomplish this, we make the *ComputeClient Runnable*:

```
package net;

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

public class ComputeClient implements
    Runnable {
    public void run() {
        try {
```

The hostname and port number will likely change, as you move the server from one platform to another. Here we simulate concurrency, using a multi-threaded arrangement:

```
        Socket s
            = new Socket("localhost",
                13);
        ObjectInputStream
            ois =
                new ObjectInputStream(
                    s.getInputStream());
        ObjectOutputStream
            oos =
                new ObjectOutputStream(
                    s.getOutputStream());
```

Submit the job for computation:

```
        oos.writeObject(new ComputeMe());
```

Get back the answer:

```
        Object o=ois.readObject();
```

Reading the object from the socket can block the current thread of execution. As a result, it may be desirable to launch the clients in a thread.

```
        System.out.println(o);
        ois.close();
        oos.close();
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

On the server side, we run a server thread that launches a new thread every time a connection is accepted on the *SocketServer* instance.

```

package net;

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

public class ComputeServer implements Runnable {
    public static void main(String args[]) {
        Thread serverThread = new Thread(new ComputeServer());
        serverThread.start();
    }
}

```

A small pause is required when starting a *SocketServer*, since the *accept* method is invoked within a thread:

```

    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
    }
    ComputeClient cc = new ComputeClient();
    cc.run();
    cc.run();
    cc.run();
}
public void run() {
    try {
        ServerSocket ss =
            new ServerSocket(13);
        while (true) {
            System.out.println("waiting");

```

A new *ComputeThread* is used to handle the computation for each new client that requests a connection:

```

        Socket socket = ss.accept();
        ComputeThread ct =
            new ComputeThread(socket);
        ct.start();
    }
}
catch (IOException e) {
    e.printStackTrace();
}
}
}

```

In summary, we have a framework for developing concurrent computation, distributed over a network of workstations. However, the implementation of the methods must be known to both the client and the server. Needing to know both of these elements, in advance, requires that the *classes be loaded* before instances of the classes are transmitted. I call this the *class configuration problem*.

## 29.10 Solving the Class Configuration Problem

This section defines and suggests solutions for the *class configuration problem*. The code in this section is experimental and not 100% reliable. This is particularly true because it has not been tested on many platforms, with different virtual machines.

When a class is loaded (even by the loader of this section) the definition verifies the byte codes before installation. Thus, all method names and signatures are valid, field names and signatures are valid, no final methods of classes are being overridden. That is all you get! A verified class can do damage. To prevent this damage you need to make an instance of the *SecurityManager* (a topic that is beyond the scope of this chapter).

We are given a client that has an implementation of a computation to be remotely computed. The implementation resides in a class file that is local to the client. When we invoke the computation on a remote server, we get a *ClassNotFoundException*. Thus, we need to find a way to provide the class files to the remote computation server, so that implementations are up to date.

There are several ways in which this can be accomplished. Since the class files are located locally, on the client's disk, we could search for them in the class path, just the way the systems' class loader does. If that fails, you can prompt the user for help locating the class file. Our approach is to locate the class in the present directory (rather than perform a search through the *zip* and *jar* files). Then if that fails, we prompt the user. The entire *RemoteClassLoader* is declared as *Serializable* so that the byte codes that define a class can be reloaded. The process of reloading a class is important because the class implementation is likely to change, while the compute server is running.

```
package net;
import java.io.*;
import java.net.*;
import java.util.*;
import futils.*;
/**
    pass an instance of this via an ObjectOutputStream
    instance to the remote host and invoke reload().
    That will redefine the class.
 */
```

The *RemoteClassLoader* is constructed on the *client*, serialized, then transmitted to the *server*. In this way, all the servers will get an update on the byte codes. This adds overhead and latency to the processing time, but it eases the configuration problem that occurs when implementations change.

```
public class RemoteClassLoader
    extends ClassLoader implements Serializable {
```

The *className* often includes a *package* name. This is generally a part of the path that is used to search for the file.

```
String className;
```

The *byteCodes* in the file are to be loaded into the class. Most of the *pain* in this program is spent in trying to locate the byte codes for a class. I have no idea why Sun does not make this easier. After all, the class is already loaded!

```
byte byteCodes[];

ClassLoader loader;

public static RemoteClassLoader
getRemoteClassLoader(Object o) {
    try {
        return new
            RemoteClassLoader(o);
    }
    catch(IOException e) {
        System.out.println("Could not make instance");
        e.printStackTrace();
        return null;
    }
}
```

The following *printPath* will show all the paths defined in the environment. A better implementation would search all the *jar* and *zip* files, looking for the class definition:

```
public static void printPath() {
    String path = System.getProperty("java.class.path");
    String sep = System.getProperty("path.separator");
    StringTokenizer st = new StringTokenizer(path, sep);
    while(st.hasMoreElements())
        System.out.println(st.nextToken());
}
// strip package name and
// add the .class suffix
```

The *getFileName* is really the heart of the program. First we substitute the “.” for the path separator, then we add the “.class” suffix. If that does not work, we ask the user for help. If the user can’t find the file, we are sunk!

```
private String getFileName() {
```

```

String sep = System.getProperty("path.separator");
if (className.indexOf(".") < 0)
    return className + ".class";
return
    gui.ReplaceString.sub(className, ".", sep)
        + ".class";
}

```

This *getClassFile* method really should be more robust. It needs to check the *jar* and *zip* files.

```

private File getClassFile() {
    File f = new File(getFileName());
    if (f.exists()) return f;
    // lets play find the file...
    printPath();
    return
        Futil.getReadFile(getFileName()
            + " not found, please help me find it!");
}
public RemoteClassLoader(Object o) throws IOException {
    Class c = o.getClass();
    className = c.getName();
    loader = c.getClassLoader();
    if (loader == null)
        System.out.println("Default system class loader");
    else
        System.out.println(loader);
    File f = getClassFile();
    System.out.println("file="+f);
    FileInputStream fis =
        new FileInputStream(f);
    byteCodes = new byte[fis.available()];
    fis.read(byteCodes);
    fis.close();
    System.out.println("#bytes="+byteCodes.length);
}

```

If you can ever find a better way to get the byte codes of a class, this is the constructor to use:

```

public RemoteClassLoader(String cn, byte bc[]) {
    className = cn;
    byteCodes = bc;
}
public void loadIt() {
    loadClass(className, true);
}

```

On the server, we invoke *reload* in order to get a fresh copy of the byte codes for defining the class.

```

public void reload() {
    // force the class to be reloaded,
    // in case it has changed.
    Class c= defineClass(

```

```

        className, byteCodes, 0, byteCodes.length);
    System.out.println("resolving:"+className);
    resolveClass(c);
}

```

Since *loadClass* is required by the super class, we define it. However, *RemoteClassLoader* instances typically don't need it. They use *reload* which forces a reloading of the byte codes.

```

protected synchronized Class loadClass(String s,
    boolean b) {
    Class cl = findLoadedClass(s);
    try {
        if (cl == null) // not in cache!
            return findSystemClass(s);
    }
    catch(ClassNotFoundException e) {
    }
    System.out.println("defining:"+s);
    Class c= defineClass(
        className, byteCodes, 0, byteCodes.length);
    System.out.println("resolving:"+className);
    if (c != null && b) resolveClass(c);
    return c;
}

public static void main(String args[]) {
    RemoteClassLoader
        rcl = RemoteClassLoader.getRemoteClassLoader(
        new ComputeMe());
    rcl.reload();
}
}

```

## 29.11 Sending E-mail using SMTP

In this section we assume that we are communicating with a mail server using SMTP (Simple Mail Transfer Protocol). This uses a communication protocol that is specified in RFC 822 <<http://sunsite.dk/RFC/rfc/rfc822.html>>. The SMTP servers typically appear on port 25 of a mail server machine. The easiest way to understand the protocol is to simulate it, by hand, using *telnet*. What follows is a record of a telnet session with a mail server. I used **bold** to indicate the part that I typed.

```

www.docjava.com{lyon}215: telnet localhost 25
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
220 www.docjava.com; ESMTP Tue, 17 Jul 2001 21:37:20 -0400
HELO LYON
250 www.docjava.com Hello localhost.localdomain [127.0.0.1],
    pleased to meet you

```

```

MAIL FROM: lyon@docjava.com
250 2.1.0 lyon@docjava.com... Sender ok
RCPT TO: lyon@docjava.com
250 2.1.5 lyon@docjava.com... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
This is some mail
.
250 2.0.0 f6I1cHU10938 Message accepted for delivery

```

As you can see, there is a very simple, text-based, protocol for sending mail to a mail server. After typing the above, I got some mail:

```

Date: Tue, 17 Jul 2001 21:38:34 -0400
From: "D. Lyon" <lyon@docjava.com>
Status:

```

```

This is some mail

```

The following code shows how to emulate the above session using sockets in a Java program:

```

package net;

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

```

The *Smtplib* class is multi-threaded, so we implement *Runnable*:

```

public class Smtplib implements Runnable {

    String recipientEmail = null;
    String senderEmail = null;
    String serverHostName = null;
    String message = null;

    int smtpPort = 25;

```

The *socket* will be used for both input from and output to the mail server:

```

Socket socket;

```

The output stream will be promoted to a *PrintWriter* instance.

```

PrintWriter pw;

```

The input stream will be promoted to a *BufferedReader* instance.

```

BufferedReader br;
InetAddress ia;
InetAddress lina;

Thread thread = new Thread(this);

```

The *emailLyon* method will send mail to me, using an IP address that is local to the *www.docjava.com* site. You will want to write your own *emailUser* method, using a local mail server:

```
public void emailLyon(String msg) {
    email(msg,
        "192.168.1.95",
        "lyon@docjava.com",
        "lyon@docjava.com");
}
```

The more general *email* method will typically be used to send mail. It is multi-threaded:

```
public void email (String msg,
    String serverHostName,
    String recipientEmail,
    String senderEmail) {
    setSenderEmail (senderEmail);
    setRecipientEmail (recipientEmail);
    setMailServerHostName (serverHostName);
    setMessage (msg);
    thread.start();
}
```

When the thread starts to *run* the message will actually be sent. Until the message is done, the parameters of the message should not be altered.

```
public void run() {
    try {
        send();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}

public void start() {
    thread.start();
}

public void send() throws IOException {
    socket =
        new Socket(serverHostName, smtpPort);
    try {
        ia = socket.getInetAddress();
        lina = ia.getLocalHost();

        pw = new
            PrintWriter(socket.getOutputStream());
        br = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()
            )
        );
    }
}
```

What follows is an implementation of the SMTP protocol:

```

        sendline("HELO " + lina.toString());
        sendline("MAIL FROM:" + senderEmail);
        sendline("RCPT TO:" + recipientEmail);
        sendline("DATA");
        sendline(message);
        sendline(".");

    } catch (Exception c) {
        socket.close();
        System.out.println (
            "Error; message send failed:\n "
            + c.getMessage());
    }
    socket.close();
}

```

After we send a line to the SMTP server, we have to read back the reply:

```

void sendline(String data) throws IOException {
    pw.println(data);
    pw.flush();
    String s = br.readLine();
    System.out.println("sendline in:" + s);
}

public void setMessage (String msg) {
    message = msg;
}

public void setSenderEmail (String email) {
    senderEmail = email;
}

public void setRecipientEmail (String email) {
    recipientEmail = email;
}

public void setMailServerHostName (String host) {
    serverHostName = host;
}

```

As you can see from the *main* the *SendMail* class is easy to use:

```

public static void main(String args[] ) {
    Smtplib sm = new Smtplib ();
    sm.emailLyon("This is a test!!");
    System.out.println(
        "Thread was started to send an email");
}

}

```

## 29.12 Summary

In this chapter we introduced the notion of using low-level, application specific protocols to perform a specific service on the Internet. The layered networking concept was introduced and a TCP/IP suite of classes were used to perform network programming.

We created simple DNS client, a web server and a simple browser, all based on the TCP/IP suite. We also showed how to create a stock quote program, able to get a list of stock quotes and a clock client, able to set the computers' clock. We also used the TCP/IP suite to send mail.

We even show how to use the TCP/IP suite and object serialization to send instances of classes across the network. This enabled a kind of remote execution that has become popular with the advent of NOWS (Networks Of Workstations). In fact, such low-cost approaches to parallel computing have become so popular that the more expensive, special-purpose parallel machine have almost totally fallen out of vogue (though this may change).

In later chapters we will show how these techniques can be extended to create a set of server-side Java programs with distributed computing abilities.

## 29.13 Exercises

1. Modify the *DateClient* class to use compressed streams (like Chapter 19 does).
2. Modify the *RemoteClassLoader* to search for classes in all the directories, *jar* files and *zip* files, in order to locate the byte codes for a class.
3. The *AutoServer* and *AutoClient* classes come with the book software. Set them up to work remotely. Alter the implementation for the *ComputableObject*. Does it work? If not, why not.
4. Alter the *RemoteClassLoader* to transfer the class file from the client to the server. Store the file in the correct path on the server. Then load the file using *Class.forName*. Does that work more reliably? Why? Hint: you will have to force the reloading of the class files. For that you will need a file class loader,

described in

<<http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/magercises/ClassLoader/solution.html>>

5. Test the *RemoteClassLoader* on two different computers with different virtual machines. Does it work reliably? Why?

6. Write a web-server that serves up the class files needed remotely. Use *java.rmi.server.RMIClassLoader(url)* to load the classes. Use serialization to provide the URL+class name to the remote class loader so that it can find the class file in question. Did you have any security problems?

7. Build your own network class loader. A sample implementation follows:

```
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;

    public Class findClass(String name) {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassData(String name) {
        // load the class data from the connection
        . . .
    }
}
```

8. Write a program to create a CSV type file of vendor addresses. How will you parse the HTML? Hint: the Swing API already has the ability to parse HTML.

The following code may help you get a start:

```
package net;
import java.util.Vector;

/**
 * VendorList allows you to save a list of
 * vendors to a file. The list comes from
 * YellowPages.com
 */
public class VendorList {
    // a Typical search URL is:
    String url = "http://www.yellowpages.com/"
        + "asp/search/Searchform.asp?"
        + "CategoryIDs=&CategoryPath="
        + "ResultStart=1&ResultCount=2000&"
        + "NameOnly=YES&TypeOnly=&SearchRelation="
        + "BName=computer&SearchMethod=&SearchMethod="
        + "&SearchMethod=&SearchBy=NAME&state=CT&City=";
    public Vector getCsv() {
```

```

    Vector raw = Browser.getUrl(url);
    return toAddressList(raw);
}
public static Vector toAddressList(Vector in) {
    Vector out = new Vector();
    //search for addresses
    for (int i=0; i < in.size();i++) {
        String s = (String)in.elementAt(i);
        if (contains(s,"Phone:"))
            out.addElement(getAddress(in,i));
    }
    return out;
}
public static boolean contains(String s1, String s2) {
    return s1.lastIndexOf(s1) != -1;
}
public static String getAddress(Vector v, int i) {
    String s = "";
    try {
        for (int j=-3; j < 0; j++) {
            s = s + "," + (String)v.elementAt(j+i);
        }
    } catch (Exception e) {}
    return s;
}
public static String vector2String(Vector raw) {
    String s = " ";
    for (int i=0; i < raw.size();i++)
        s = s + raw.elementAt(i);
    return s;
}
public static void main(String args[]) {
    VendorList vl = new VendorList();
    Browser.print(vl.getCsv());
}
}
}

```

9. Write a program that gets the byte codes for a class by using the *Class.getResourceAsStream* method. Pass the class name in by using the *Class.getName* method. For example:

```

public static byte[] getByteCodes(Object o) throws
    IOException {
    Class c = o.getClass();
    InputStream is = c.getResourceAsStream(c.getName());
    //... add more code here
}

```

How else can you get the byte codes for an *Object*?

10. Write a program that will parse the return output and place it into classes that break the values down into primitive data types. Using a *StringTokenizer* that uses comma and quote as delimiters. For example:

```
StringTokenizer t = new StringTokenizer(line, "\\");
System.out.println(line);

String symbol = t.nextToken();
double price = Double.valueOf(t.nextToken()).doubleValue();

t.nextToken();
Date date = new Date();
t.nextToken();
Date lastTrade = date;

double change = Double.valueOf(t.nextToken()).doubleValue();
double open = Double.valueOf(t.nextToken()).doubleValue();
double bid = Double.valueOf(t.nextToken()).doubleValue();
double ask = Double.valueOf(t.nextToken()).doubleValue();
int volume = Integer.valueOf(t.nextToken()).intValue();
```

Have your program return a list of these quotes.

11. Modify the program in exercise 10 so that the quotes scroll across the screen on a ticker.
12. Modify the program in exercise 11 to add a swing interface so that people can add symbols to their stock listing and save them to a file.