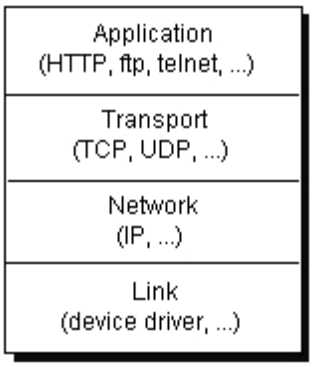
Unit-5 Network Programming

Networking Basics

Computers running on the Internet communicate to each other using either the **Transmission Control Protocol (TCP)** or the **User Datagram Protocol (UDP),** as this diagram illustrates:



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the **java.net package**. These classes provide **system-independent network communication**. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

Transmission Control Protocol (TCP)

TCP (Transmission Control Protocol) is a connection-based protocol that provides a reliable flow of data between two computers.

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. If you want to speak to your friend, a connection is established when you dial his phone number and he answers. You send data back and forth over the connection by speaking to one another over the phone lines. Like the phone company, **TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.**

TCP provides a point-to-point channel for applications that require reliable communications. The **Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet** are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, user end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

User Datagram Protocol (UDP)

UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called **datagrams**, from one computer to another with no guarantees about arrival.

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called datagrams, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is

independent of any other.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, **they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.**

Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.

Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.

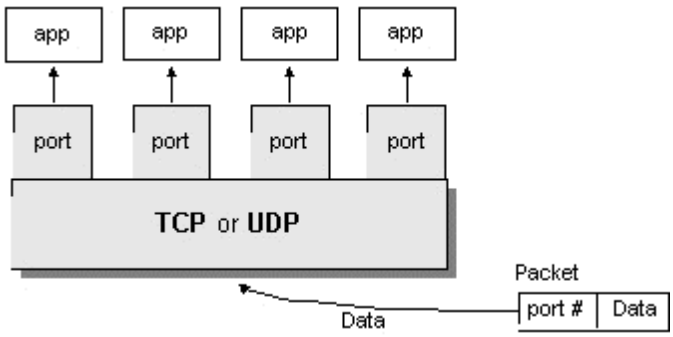
Many firewalls and routers have been configured not to allow UDP packets. If you're having trouble connecting to a service outside your firewall, or if clients are having trouble connecting to your service, you should check whether UDP is permitted.

Ports

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of ports.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its **32-bit IP** address, which IP uses to deliver data to the right computer on the network. **Ports** are identified by a **16-bit number**, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port.
In datagram-based communication such as UDP, the datagram packet contains the port number of its destination and UDP routes the packet to the appropriate application.



Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called well-known ports. Your applications should not attempt to bind to them.

Networking Classes in the JDK

Through the classes in **java.net**, Java programs can use TCP or UDP to communicate over the Internet. The **URL, URLConnection, Socket, and ServerSocket classes** all **use TCP** to communicate over the network. The **DatagramPacket, DatagramSocket, and MulticastSocket classes** are for use with **UDP**.

Working with URLs

URL is the acronym for **Uniform Resource Locator**. It is a reference (an address) to a resource on the Internet. You provide URLs to your favorite Web browser so that it can locate files on the Internet in the same way that you provide addresses on letters so that the post office can locate your correspondents.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. **Java programs can use a class called URL in the java.net package to represent a URL address.**

The term URL can be ambiguous. It can refer to an Internet address or a URL object in a Java program. Here "URL address" is used to mean an Internet address and "URL object" to refer to an instance of the URL class in a program.

URL

URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet.If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

A URL has **two main components**:

Protocol identifier: For the URL http://example.com, the **protocol identifier is http.**

Resource name: For the URL http://example.com, the **resource name is example.com.**

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, **the resource name contains one or more of the following components:**

Host Name

The name of the machine on which the resource lives.

Filename

The pathname to the file on the machine.

Port Number

The port number to which to connect (typically optional).

Reference

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Creating a URL

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

```
URL myURL = new URL("http://example.com/");
```

The URL object created above represents **an absolute URL**. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a relative URL address.

Creating a URL Relative to Another

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:
http://example.com/pages/page1.html
http://example.com/pages/page2.html
You can create URL objects for these pages relative to their common base URL:
http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:

URL(URL baseURL, String relativeURL)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If baseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

Other URL Constructors

new URL("http", "example.com", "/pages/page1.html");
This is equivalent to
new URL("http://example.com/pages/page1.html");

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.

The final URL constructor adds the port number to the list of arguments used in the previous constructor:
URL url = new URL("http", "example.com", 80, "pages/page1.html");
This creates a URL object for the following URL:
http://example.com:80/pages/page1.html
If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method or the equivalent toExternalForm method.

URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:
http://example.com/hello world/
To make these characters legal they need to be encoded before passing them to the URL constructor.
URL url = new URL("http://example.com/hello%20world");
Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the java.net.URI class to automatically take care of the encoding for you.
URI uri = new URI("http", "example.com", "/hello world/", "");
And then convert the URI to a URL.
URL url = uri.toURL();

MalformedURLException

Each of the four URL constructors throws a MalformedURLException if the arguments to the constructor refer to a null or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

```
try {
   URL myURL = new URL(...);
}
```

```
catch (MalformedURLException e) {
   // exception handler code here
   // ...
}
```

Parsing a URL

The URL class provides several methods that let you query URL objects. You can get the **protocol, authority, host name, port number, path, query, filename, and reference** from a URL using these accessor methods:

getProtocol
Returns the protocol identifier component of the URL.

getAuthority
Returns the authority component of the URL.

getHost
Returns the host name component of the URL.

getPort
Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.

getPath
Returns the path component of this URL.

getQuery
Returns the query component of this URL.

getFile
Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.

getRef
Returns the reference component of the URL.

Note:
Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric.

You can use these getXXX methods to get information about the URL regardless of the constructor that you used to create the URL object.

The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need. This small example program creates a URL from a string specification and then uses the URL object's accessor methods to parse the URL:

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

public class ParseURL {
   public static void main(String[] args) throws Exception {
```

```
        URL aURL = new URL("http://example.com:80/docs/books/tutorial"
                    + "/index.html?name=networking#DOWNLOADING");

        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("authority = " + aURL.getAuthority());
        System.out.println("host = " + aURL.getHost());
        System.out.println("port = " + aURL.getPort());
        System.out.println("path = " + aURL.getPath());
        System.out.println("query = " + aURL.getQuery());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

Here is the output displayed by the program:

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

Reading Directly from a URL

After you've successfully created a URL, you can call the URL's **openStream() method** to get a stream from which you can read the contents of the URL. **The openStream() method returns a java.io.InputStream object**, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.google.com.np/. It then opens a **BufferedReader** on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream:

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

public class URLReader {
    public static void main(String[] args) throws Exception {

        URL oracle = new URL("http://www.google.com.np/");
        BufferedReader in = new BufferedReader(
        new InputStreamReader(oracle.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
```

```
    }
}
```

When you run the program, you should see, scrolling by in your command window, the HTML commands and textual content from the HTML file located at http://www.google.com.np/.

Connecting to a URL

After you've successfully created a URL object, you can call the URL object's **openConnection** method to get a URLConnection object, or one of its protocol specific subclasses, e.g. **java.net.HttpURLConnection**

You can use this URLConnection object to setup parameters and general request properties that you may need before connecting. **Connection to the remote object represented by the URL is only initiated when the URLConnection.connect method is called**. When you do this you are initializing a communication link between your Java program and the URL over the network. For example, the following code opens a connection to the site example.com:

```
try {
    URL myURL = new URL("http://example.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) {
    // new URL() failed
    // ...
}
catch (IOException e) {
    // openConnection() failed
    // ...
}
```

A new URLConnection object is created every time by calling the openConnection method of the protocol handler for this URL.

You are not always required to explicitly call the connect method to initiate the connection. Operations that depend on being connected, like getInputStream, getOutputStream, etc, will implicitly perform the connection, if necessary.

Now that you've successfully connected to your URL, you can use the URLConnection object to perform actions such as reading from or writing to the connection. The next example shows how.

Reading from a URLConnection

The following program performs the same function as the URLReader program shown in Reading Directly from a URL.
However, rather than getting an input stream directly from the URL, this program explicitly retrieves a URLConnection object and gets an input stream from the connection. The connection is opened implicitly by calling getInputStream. Then, like URLReader, this program creates a BufferedReader on the input stream and reads from it.

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

```
import java.io.*;

public class URLConnectionReader {
   public static void main(String[] args) throws Exception {
      URL oracle = new URL("http://www.oracle.com/");
      URLConnection yc = oracle.openConnection();
      BufferedReader in = new BufferedReader(new InputStreamReader(
                              yc.getInputStream()));
      String inputLine;
      while ((inputLine = in.readLine()) != null)
         System.out.println(inputLine);
      in.close();
   }
}
```

The output from this program is identical to the output from the program that opens a stream directly from the URL. You can use either way to read from a URL. However, reading from a URLConnection instead of reading directly from a URL might be more useful. This is because you can use the URLConnection object for other tasks (like writing to the URL) at the same time.

Sockets

URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

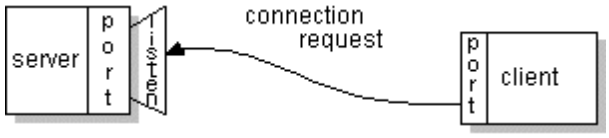
In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

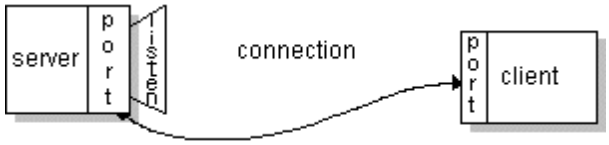
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The **java.net package** in the Java platform provides a **class, Socket**, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the **ServerSocket** class, which implements a socket that **servers** can use to listen for and accept connections to clients.

If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation.

Establishing a Simple Server Using Stream Sockets

Establishing a simple server in Java requires five steps.

Step 1: Create a ServerSocket

First step is to create a **ServerSocket object**. A call to the ServerSocket constructor, such as

ServerSocket server = new ServerSocket(portNumber, queueLength);

registers an available TCP port number and specifies themaximum number of clients that can wait to connect to the server (i.e., the queue length). The port number is used by clients to locate the server application on the server computer. This is often called the handshake point. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients—a process known as **binding the server to the port**. Each client will ask to connect to the server on this port. Only one application at a time can be bound to a specific port on the server.

Step 2: Wait for a Connection

Programs manage each client connection with a Socket object. In Step 2, the server listens indefinitely (or blocks) for an attempt by a client to connect. To listen for a client connection, the program calls ServerSocket method accept, as in

Socket connection = server.accept();

which returns a Socket when a connection with a client is established. The Socket allows the server to interact with the client. The interactions with the client actually occur at a different server port from the handshake point. This allows the port specified in Step 1 to be used again in a multithreaded server to accept another client connection.

Step 3: Get the Socket's I/O Streams

Step 3 is to get the **OutputStream** and **InputStream** objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to the client via an OutputStream and receives information from the client via an InputStream. The server invokes method **getOutputStream** on the Socket to get a reference to the Socket's OutputStream and invokes method **getInputStream** on the Socket to get a reference to the Socket's InputStream.

```
Socket con=new Socket("localHost",95);
BufferedReader in=new BufferedReader(new InputStreamReader(con.getInputStream()));
PrintWriter out=new PrintWriter(con.getOutputStream(),true);
```

The beauty of establishing these relationships is that whatever the server writes to the PrintWriter is sent via the OutputStream and is available at the client's InputStream, and whatever the client writes to its OutputStream (with a corresponding PrintWriter) is available via the server's InputStream. The transmission of the data over the network is seamless and is handled completely by Java.

Step 4: Perform the Processing

In which the server and the client communicate via the OutputStream and InputStream objects.

Step 5: Close the Connection

when the transmission is complete, the server closes the connection by invoking the close method on the streams and on the Socket.

```
in.close();
out.close();
con.close();
```

Establishing a Simple Client Using Stream Sockets

Establishing a simple client in Java requires four steps.

Step 1: Create a Socket to Connect to the Server

In first step we create a Socket to connect to the server. The Socket constructor establishes the connection. For example, the statement

Socket connection = new Socket(serverAddress, port);

uses the Socket constructor with two arguments—**the server's address (serverAddress)** and the **port number**. If the connection attempt is successful, this statement returns a Socket. A connection attempt that fails throws an instance of a subclass of IOException, so many programs simply catch IOException. An UnknownHostException occurs specifically when the system is unable to resolve the server name specified in the call to the Socket constructor to a corresponding IP address.

Step 2: Get the Socket's I/O Streams

Here the client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream as described earlier.

Step 3: Perform the Processing

In this phase the client and the server communicate via the InputStream and OutputStream objects.

Step 4: Close the Connection

In Step 4, the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket as described earlier.

InetAddress class

Usually, you don't have to worry too much about Internet addresses, the numerical host addresses that consist of four bytes (or, with IPv6, 16 bytes) such as 132.163.4.102. However, you can use the **InetAddress** class if you need to convert between host names and Internet addresses.

As of JDK 1.4, the **java.net package** supports IPv6 Internet addresses, provided the host operating system does.

The static **getByName** method returns an InetAddress object of a host. For example,

InetAddress address = InetAddress.getByName("HostName");

returns an InetAddress object that encapsulates the sequence of four bytes such as 132.163.4.104.

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example,the host name java.sun.com corresponds to three different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the **getAllByName** method.

InetAddress[] addresses = InetAddress.getAllByName(host);

String getHostAddress()-returns a string with decimal numbers, separated by periods, for example, "132.163.4.102".

String getHostName()-returns the host name.

Program for chatting between client and server

```
//Server.java
import java.io.*;
import java.net.*;
public class Server
```

```
{
public static void main(String a[])throws IOException
{
try
{
System.out.println("SERVER:.....\n");
ServerSocket s=new ServerSocket(95);
System.out.println("Server Waiting For The Client");
Socket cs=s.accept();
InetAddress ia=cs.getInetAddress();
String cli=ia.getHostAddress();
System.out.println("Connected to the client with IP:"+cli);
BufferedReader in=new BufferedReader(new
InputStreamReader(cs.getInputStream()));
PrintWriter out=new PrintWriter(cs.getOutputStream(),true);
do
{
BufferedReader din=new BufferedReader(new
InputStreamReader(System.in));
System.out.print("To Client:");
String tocl=din.readLine();
out.println(tocl);
String st=in.readLine();
if(st.equalsIgnoreCase("Bye")||st==null)break;
System.out.println("From Client:"+st);
}while(true);
in.close();
out.close();
cs.close();
}
catch(IOException e) { }
}
}
```

//Client.java

```
import java.io.*;
import java.net.*;
public class Client
{
public static void main(String a[])throws IOException
{
try
```

```
{
System.out.println("CLIENT:.....\n");
Socket con=new Socket("localHost",95);
BufferedReader in=new BufferedReader(new
InputStreamReader(con.getInputStream()));
PrintWriter out=new PrintWriter(con.getOutputStream(),true);
while(true)
{
String s1=in.readLine();
System.out.println("From Server:"+s1);
System.out.print("Enter the messages to the server:");
BufferedReader din=new BufferedReader(new
InputStreamReader(System.in));
String st=din.readLine();
out.println(st);
if(st.equalsIgnoreCase("Bye")||st==null)break;
}
in.close();
out.close();
con.close();
}
catch(UnknownHostException e){ }
}
}
```

Serving Multiple Clients

There is one problem with the simple server in the preceding example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to use the server at the same time. Rejecting multiple connections allows any one client to monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection, that is, when the call to accept was successful, we will launch a new thread to take care of the connection between the server and that client. The main program will just go back and wait for the next connection. For this to happen, the main loop of the server should look like this:

```
while (true)
{
  Socket incoming = s.accept();
  Runnable r = new ThreadedEchoHandler(incoming);
```

```
  Thread t = new Thread(r);
  t.start();
}
```

The THReadedEchoHandler class implements Runnable and contains the communication loop with the client in its run method.

```
class ThreadedEchoHandler implements Runnable
{ . . .
  public void run()
  {
    try
    {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      ...process input and send response...
      incoming.close();
    }
    catch(IOException e)
    {
      handle exception
    }
  }
}
```

Because each connection starts a new thread, multiple clients can connect to the server at the same time.

Socket Timeouts

In real-life programs, you don't just want to read from a socket, because the read methods will block until data are available. If the host is unreachable, then your application waits for a long time and you are at the mercy of the underlying operating system to time out eventually. Instead, you should decide what timeout value is reasonable for your particular application. Then, call the **setSoTimeout** method to set a timeout value (in milliseconds).

```
Socket s = new Socket(. . .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, then all subsequent read and write operations throw a **SocketTimeoutException** when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try
{
  Scanner in = new Scanner(s.getInputStream());
  String line = in.nextLine();
```

```
  . . .
}
catch (InterruptedIOException exception)
{
  react to timeout
}
```

Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling interrupt.

To interrupt a socket operation, you use a **SocketChannel**, a feature of the **java.nio package.** Open the SocketChannel like this:

SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));

A channel does not have associated streams. Instead, it has read and write methods that make use of Buffer objects.These methods are declared in interfaces **ReadableByteChannel** and **WritableByteChannel**.

If you don't want to deal with buffers, you can use the Scanner class to read from a SocketChannel because Scanner has a constructor with a ReadableByteChannel parameter:

Scanner in = new Scanner(channel);

To turn a channel into an output stream, use the static Channels.newOutputStream method.

OutputStream outStream = Channels.newOutputStream(channel);

That's all you need to do. **Whenever a thread is interrupted during an open, read, or write operation, the operation does not block but is terminated with an exception.**

Half-Close

When a client program sends a request to the server, the server needs to be able to determine when the end of the request occurs. For that reason, many Internet protocols (such as SMTP) are line oriented. Other protocols contain a header that specifies the size of the request data. Otherwise, indicating the end of the request data is harder than writing data to a file. With a file, you'd just close the file at the end of the data. However, if you close a socket, then you immediately disconnect from the server.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the request data, but keep the input stream open so that you can read the response.

The client side looks like this:

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// send request data
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// now socket is half closed
// read response data
while (in.hasNextLine()) != null) { String line = in.nextLine(); . . . }
socket.close();
```

The server side simply reads input until the end of the input stream is reached.This protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

Sending E-Mail via javax.mail API

```
//SendMail.java
import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class SendMail {

	public static void main(String[] args) {

		final String username = "knavin12@gmail.com";
		final String password = "password";

		Properties props = new Properties();
		props.put("mail.smtp.auth", "true");
		props.put("mail.smtp.starttls.enable", "true");
		props.put("mail.smtp.host", "smtp.gmail.com");
		props.put("mail.smtp.port", "587");
```

```
		Session session = Session.getInstance(props,
		  new javax.mail.Authenticator() {
			protected PasswordAuthentication getPasswordAuthentication() {
				return new PasswordAuthentication(username, password);
			}
		  });

		try {

			Message message = new MimeMessage(session);
			message.setFrom(new InternetAddress("knavin12@gmail.com"));
			message.setRecipients(Message.RecipientType.TO,
				InternetAddress.parse("knavin12@gmail.com"));
			message.setSubject("Testing Subject");
			message.setText("Hello Navin,"
				+ "\n\n Congrates u succeded sending mail\n through Java.mail
				API.");

			Transport.send(message);

			System.out.println("Your email has been sent successfully");

		} catch (MessagingException e) {
			throw new RuntimeException(e);
		}
	}
}
```