UNIT - VI

NETWORKING

BASICS OF NETWORK PROGRAMMING:

Network: Inter Connection of computers is called a network.

In a network, there may be several computers, some of them receiving the services and some of them providing the services to others. The computer which receives service is called a client and the computer which provides the service is called a server. Remember a client sometimes acts as a server and a server acts as a client.

There are 3 requirements to establish a network:

- Hardware: includes the computers, cables, modems, hubs etc.
- Software: includes programs to communicate between servers and clients.
- Protocol: set of rules and regulations that represents a way to establish connection and helps in sending and receiving data in a standard format. Popular protocol suit is TCP/IP. TCP is connection oriented and IP connection less potocols.

e.g.: HTTP, FTP, SMTP, POP, UDP etc.

IP address: An IP address is a unique identification number alloted to every computer on a network or internet. IP address contains some bytes which identify the network and the actual computer inside the network.

DNS: Domain Naming Service is a service on internet that maps the IP addresses with corresponding website names.

Socket Programming: is a low level way to establish connection between server and a client with the help of a Socket.

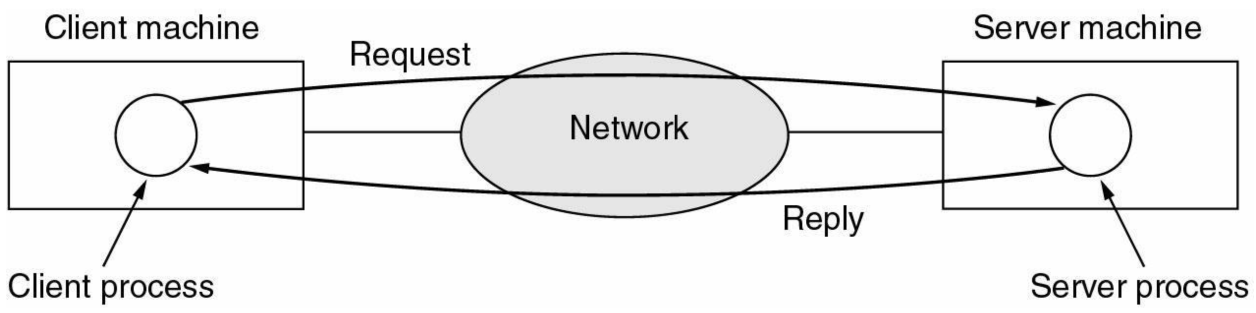


Figure 1. The client-server models involves requests and replies: Application- server

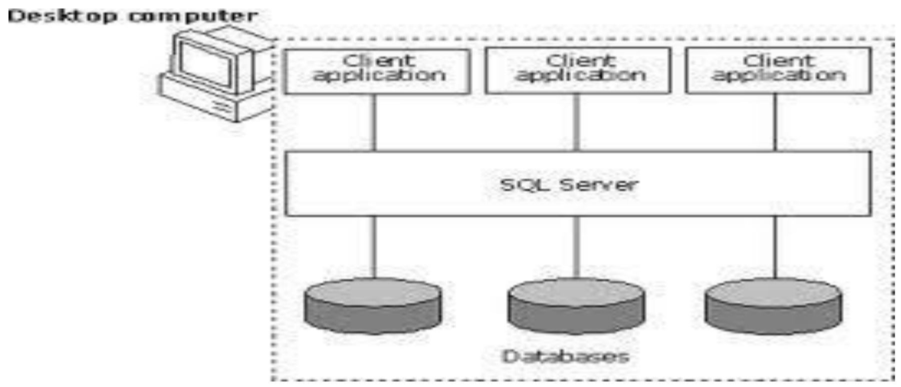


Figure 2. The client-server models involves requests and replies: Application- server(Sql)

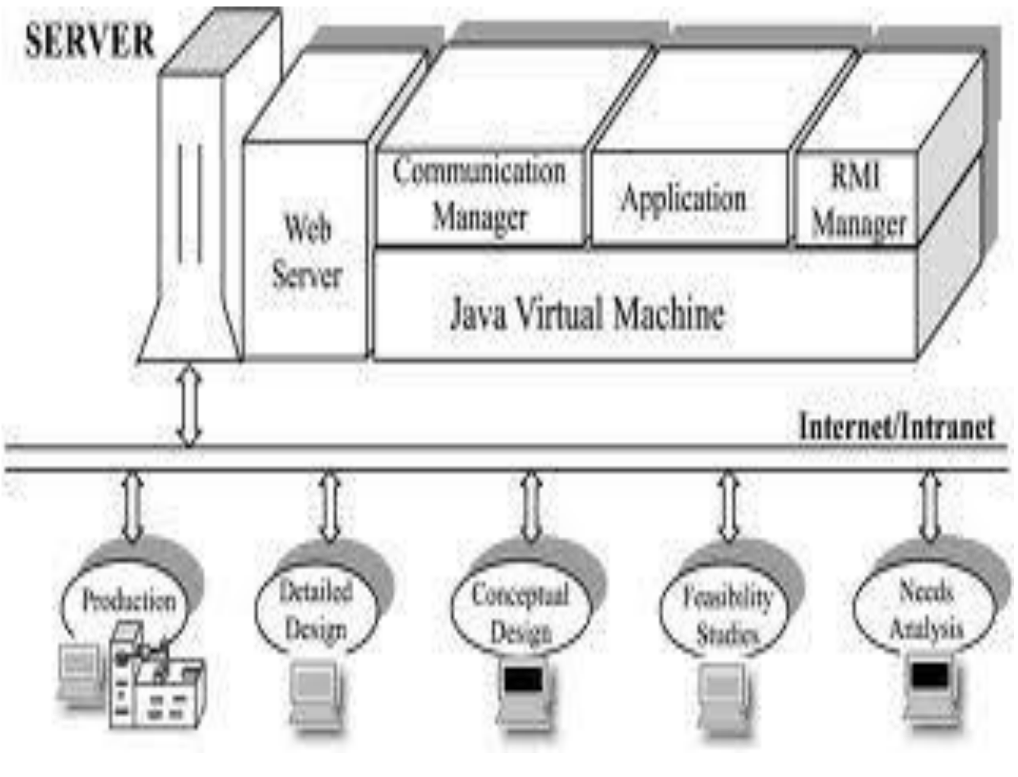


Figure 3. The client-server models involves requests and replies: Client-Server (Java)

One of Java's great strengths is painless networking. As much as possible, the underlying details of networking have been abstracted away. The programming model we use is that of the file programming model. In addition, Java's built-in multithreading is exceptionally handy when dealing with another networking issue: handling multiple connections at once. The java.net package includes classes that enable the programmer to easily pass data across networks.

Before plowing through the examples in the next several lessons, you should have an understanding of some networking basics. Also, to boost your confidence, we've included a section that reviews what you may already know about networking in Java without even realizing it.

- Sending E-Mails
- Online shoping
- Browsing
- Chating
- Downloading files
- Online meeting
- Data storage
- Downloading remote applets

Getting Hands-on Java Networking example

1. **Find Your Host Name/IP Address**
   In this section you will learn about the **getLocalHost()** method to print the Host name as well as the IP Address of the local system. For do for the same we have to call **InetAddress** class then we need to create a object, in which we store the local host information.
2. **Find Your Host Name**
   Here we are going to explore a method to retrieve the host name of the local system in a very simple example. Here we are just call the InetAddress class and after made a object of that class we call a **getLocalHost()** method and pass it in the object.
3. **Getting list of Local Interfaces on a machine**
   Here we are going to explain the method to find out the total no of list of local interfaces available on a machine. Here we are give a complete example named URLDemo.java.
4. **Getting the Local port**
   In this section you will learn that how a user can access the Local port. Here we are going to make it easier to understand the step be step process by the complete example on this topic.
5. **Find the Host name in reverse of given IP address**
   In this section you will learn to retrieve the information about the local host name using the **getHostName()** method. Here is an example that provides the usage of the **getHostName()** method in more detail.

6. **How to retrieve URL information**
   Here we are going to explain a method for retrieve all the network information for a given URL. In the given below example we use the URL class and make a object.
7. **Convert URI to URL**
   Here we are going to explain the method to change Uniform Resource Identifier (URI) reference to uniform resource locator (URL). Here we are give a complete example named "ConvertURItoURL.java".
8. **To retrieve the IP address from Host Name, vice-versa**
   Here we are going to explain the method to find out the IP address from host name and to vice verse. Here we are give a complete example named **HostLookup.java**. In which we call a InetAddress and make a object and pass the input value in it.
9. **Construct a DatagramPacket to receive data**
   In this section we provide a complete code of a example based on the method **DatagramPacket(buffer, buffer.length)** for constructs a DatagramPacket for receiving packets of length **length** in more generic way.
10. **Construct a DatagramSocket on an unspecified port**
    In this section we are going to explain the process to construct a **DatagramPacket** object in more generic way. Here we provide a complete example based on the method for creating the **DatagramSocket** object via **DatagramSocket().**
11. **Return the MIME Header**
    In this section you will learn about the method to retrieve the MIME header. MIME is stands for Multipurpose Internet Mail Extensions which is an Internet Standard that extends the format of e-mail to support. A container for MimeHeader objects, which represent the MIME headers that is present in a MIME part of data.
12. **Open a URLConnection to specific website address**
    Here we are going to establish a connection to a specific web address via a complete example. In the example we create a class getURLConnection and initialize a variable of url to URL and ucon of URLConnection.
13. **Print the URL of a URLConnection**
    In this section we are going to describe, how to retrieve the value of the URL assign to the url object. Here is the complete code of the program in which first of all we establish a connection then display it.
14. **Getting Image from a URL**
    This example shows how to get an image from the given URL.
15. **URLInformation**
    Here we are going to explain the method to find out the URL information. This program defines the IOException for the exception handling. This exception is thrown to indicate an I/O problem.
16. **GetHTTPHeader**
    In this section, you will learn how to get **content-length, content- type**, and **last-modify date** of a file. Here we provide a complete example that uses **getHeaderFieldKey()** method**.**
17. **EncoderTest**
    This is a simple program of java network and supports the **java.net** package. Here, we have defined the class named **EncoderTest.java**. This class contains static methods for converting a String to the application-urlencoded MIME format.

18. **EchoClientSocket**
    In this section, we are going to explain the method to find out the hostname port number of a local machine. This program defines the IOException for the exception handling.
19. **Client Socket Information**
    In this section, you will learn how to get client socket information. This is a very simple program of java network. Here, we have used the class named **ClientSocketInformation.java** that implement the constructor of the **Socket** class passing two arguments as **hostName** and **TIME_PORT**.
20. **ReadIPAddress**
    Here, we are going to explore a method to retrieve the IPAddress of the local system in a very simple example. For do for the same we have to call **InetAddress** class then we need to create a object, in which we store the local host information.
21. **Local Port Scanner**
    This is a simple program of java network. Here, we are going to define class named "**LocalPortScanner**" in which we use ServerSocket class of **java.net** package which takes local port number as argument.
22. **Low port Scanner**
    In this section, you will learn how to get local port number of the local machine. Here, we define the name of program **LowPortScanner**. In this we check the argument and create a socket object.
23. **URL Connection Reader**
    In this section, you will learn about the url connection reader. Here, first of all we are going to define class named "**URLConnectionReader**". After that we make a URL object passing a url "**http://www.javajazzup.com**" as string in its constructor.
24. **URL Reader File**
    This is a simple program of java network. In this section, you will learn how to read HTML file. Here, we create a URL object and pass a url. After that, we call the URL?s **openStream()** method to get a stream from which the contents of the URL can be read.
25. **RMI Client And RMI Server Implementation**
    The RMI application comprises of the two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.

Java .net package:

Provides the classes for implementing networking applications. Using the socket classes, you can communicate with any server on the Internet or implement your own Internet server. A number of classes are provided to make it convenient to use Universal Resource Locators (URLs) to retrieve data on the Internet.Addresses are used throughout the java.net APIs as either host identifiers, or socket endpoint identifier. **The java.net package can be roughly divided in two sections:**

- *A Low Level API*, which deals with the following abstractions:
    - *Addresses*, which are networking identifiers, like IP addresses.
    - *Sockets*, which are basic bidirectional data communication mechanisms.
    - *Interfaces*, which describe network interfaces.

- *A High Level API*, which deals with the following abstractions:
    - *URIs*, which represent Universal Resource Identifiers.
    - *URLs*, which represent Universal Resource Locators.
    - *Connections*, which represents connections to the resource pointed to by *URLs*.

Provides the classes for implementing networking applications.

Interface Summary	
ContentHandlerFactory	This interface defines a factory for content handlers.
DatagramSocketImplFactory	This interface defines a factory for datagram socket implementations.
FileNameMap	A simple interface which provides a mechanism to map between a file name and a MIME type string.
SocketImplFactory	This interface defines a factory for socket implementations.
SocketOptions	Interface of methods to get/set socket options.
URLStreamHandlerFactory	This interface defines a factory for URL stream protocol handlers.

Class Summary	
Authenticator	The class Authenticator represents an object that knows how to obtain authentication for a network connection.
CacheRequest	Represents channels for storing resources in the ResponseCache.
CacheResponse	Represent channels for retrieving resources from the ResponseCache.
ContentHandler	The abstract class ContentHandler is the superclass of all classes that read an Object from a URLConnection.
CookieHandler	A CookieHandler object provides a callback mechanism to hook up a HTTP state management policy implementation into the HTTP

	protocol handler.
DatagramPacket	This class represents a datagram packet.
DatagramSocket	This class represents a socket for sending and receiving datagram packets.
DatagramSocketImpl	Abstract datagram and multicast socket implementation base class.
HttpURLConnection	A URLConnection with support for HTTP-specific features.
Inet4Address	This class represents an Internet Protocol version 4 (IPv4) address.
Inet6Address	This class represents an Internet Protocol version 6 (IPv6) address.
InetAddress	This class represents an Internet Protocol (IP) address.
InetSocketAddress	This class implements an IP Socket Address (IP address + port number) It can also be a pair (hostname + port number), in which case an attempt will be made to resolve the hostname.
JarURLConnection	A URL Connection to a Java ARchive (JAR) file or an entry in a JAR file.
MulticastSocket	The multicast datagram socket class is useful for sending and receiving IP multicast packets.
NetPermission	This class is for various network permissions.
NetworkInterface	This class represents a Network Interface made up of a name, and a list of IP addresses assigned to this interface.
PasswordAuthentication	The class PasswordAuthentication is a data holder that is used by Authenticator.
Proxy	This class represents a proxy setting, typically a type (http, socks) and a socket address.
ProxySelector	Selects the proxy server to use, if any, when connecting to the network resource referenced by a URL.
ResponseCache	Represents implementations of URLConnection caches.

SecureCacheResponse	Represents a cache response originally retrieved through secure means, such as TLS.
ServerSocket	This class implements server sockets.
Socket	This class implements client sockets (also called just "sockets").
SocketAddress	This class represents a Socket Address with no protocol attachment.
SocketImpl	The abstract class SocketImpl is a common superclass of all classes that actually implement sockets.
SocketPermission	This class represents access to a network via sockets.
URI	Represents a Uniform Resource Identifier (URI) reference.
URL	Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
URLClassLoader	This class loader is used to load classes and resources from a search path of URLs referring to both JAR files and directories.
URLConnection	The abstract class URLConnection is the superclass of all classes that represent a communications link between the application and a URL.
URLDecoder	Utility class for HTML form decoding.
URLEncoder	Utility class for HTML form encoding.
URLStreamHandler	The abstract class URLStreamHandler is the common superclass for all stream protocol handlers.

Enum Summary	
Authenticator.RequestorType	The type of the entity requesting authentication.
Proxy.Type	Represents the proxy type.

Exception Summary	
BindException	Signals that an error occurred while attempting to bind a socket to a local address and port.
ConnectException	Signals that an error occurred while attempting to connect a socket to a remote address and port.
HttpRetryException	Thrown to indicate that a HTTP request needs to be retried but cannot be retried automatically, due to streaming mode being enabled.
MalformedURLException	Thrown to indicate that a malformed URL has occurred.
NoRouteToHostException	Signals that an error occurred while attempting to connect a socket to a remote address and port.
PortUnreachableException	Signals that an ICMP Port Unreachable message has been received on a connected datagram.
ProtocolException	Thrown to indicate that there is an error in the underlying protocol, such as a TCP error.
SocketException	Thrown to indicate that there is an error in the underlying protocol, such as a TCP error.
SocketTimeoutException	Signals that a timeout has occurred on a socket read or accept.
UnknownHostException	Thrown to indicate that the IP address of a host could not be determined.
UnknownServiceException	Thrown to indicate that an unknown service exception has occurred.
URISyntaxException	Checked exception thrown to indicate that a string could not be parsed as a URI reference.

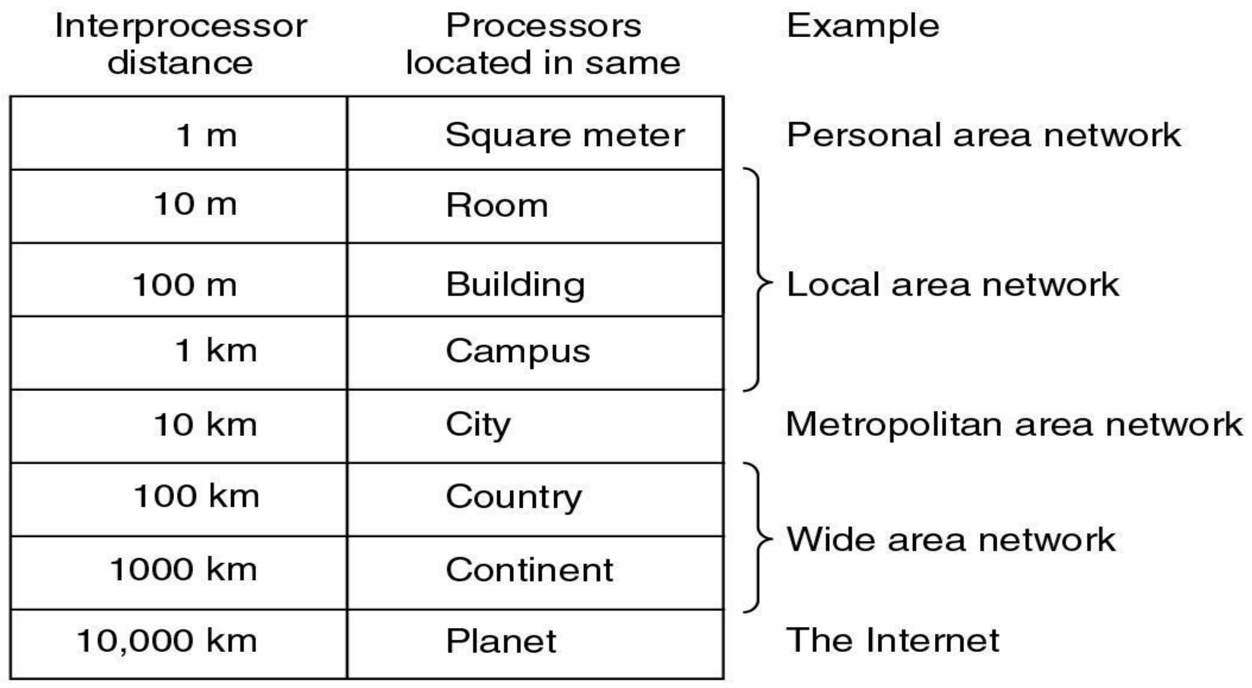
What Is the Internet and How Does It Work?

Asking the question What is the Internet? may bring about a heated discussion in some circles. In this book, the *Internet* is defined as the collection of all computers that are able to communicate, using the Internet protocol suite, with the computers and networks registered with the *Internet Network Information Center* (InterNIC). This definition includes all computers to which you can directly (or indirectly through a firewall) send Internet Protocol packets.

Computers on the Internet communicate by exchanging packets of data, known as Internet Protocol, or IP, packets. IP is the network protocol used to send information from one computer to another over the Internet. All computers on the Internet (by our definition in this book) communicate using IP. IP moves information contained in IP packets. The IP packets are routed via special routing algorithms from a source computer that sends the packets to a destination computer that receives them. The routing algorithms figure out the best way to send the packets from source to destination.

In order for IP to send packets from a source computer to a destination computer, it must have some way of identifying these computers. All computers on the Internet are identified using one or more IP addresses. A computer may have more than one IP address if it has more than one interface to computers that are connected to the Internet.

Classification of interconnected processors by scale:



Interprocessor distance	Processors located in same	Example
1 m	Square meter	Personal area network
10 m	Room	Local area network
100 m	Building	
1 km	Campus	
10 km	City	Metropolitan area network
100 km	Country	Wide area network
1000 km	Continent	
10,000 km	Planet	The Internet

JDK 1.0). Since version 1.4 non-blocking IO through buffers and channels is also integral part of standard Java distribution (but we are not going to go through non-blocking IO in this tutorial).

Package java.net can be divided into two parts: application layer and transport layer classes. TCP and UDP protocols (on top of IP) are fully supported so a complete freedom while writing network applications is guaranteed.

Application layer classes

This part includes higher level classes which enables working with URI and URL addresses and connections created using those addresses. The most important classes are URI, URL, URLConnection, and HttpURLConnection. These classes can be useful if for acquiring connections with a computer on Internet, if you want to parse HTML pages, create HTTP-based connections etc.

This part of the java.net package is all that is needed for a basic support for application layer in TCP/IP protocol â€“ there is a huge number of third party classes on Internet which are not integral part of the standard Java distribution â€“ some of them will help you send an email, some other will help you to contact some Web service etc.

Transport layer classes

This part of the java.net package is what distributed network programming is all about. This is the lowest level that Java allows somebody to work on while doing network programming (TCP & UDP) and what all other high level network and distributed programming Java toolkits are based on. Basic three elements in this below level part of java.net are:

- address
- socket
- interface

Address (IPv4, Ipv6) is being used as a identifier for network adapter in a network. Although two different classes exist in this package for IPv4 and IPv6 (INet4Address and INet6Address), a suggestion is to use base class InetAddress because Java will automatically adopt and use the right class so a programmer does not have to worry about protocol version.

Interfaces are being used for querying network adapter and hardware for properties (through Java interface NetworkInterface). Some of things that can be found out are which is hardware (MAC) address of the network adapter, what is device MTU, which addresses are mapped on a network device etc.

Socket is, basically, standardized bidirectional communication mechanism. Socket class for TCP is Socket and socket class for UDP protocol called DatagramSocket. Both sides in UDP communication are with same rights (client/server paradigm is not completely meaningful in this case).

UDP protocol is much simpler and usually being used for streaming protocols (audio & video network transmissions) because it is unsafe transportation mechanism. Being unsafe means that it is not a good choice in network programming so usually it's best to choose TCP as a network transportation protocol.

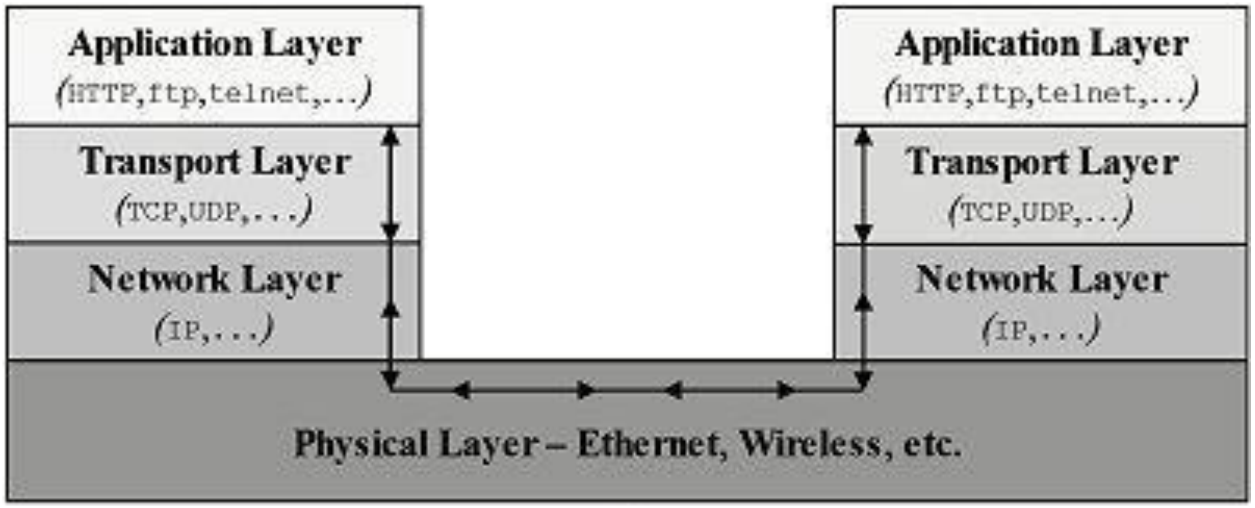
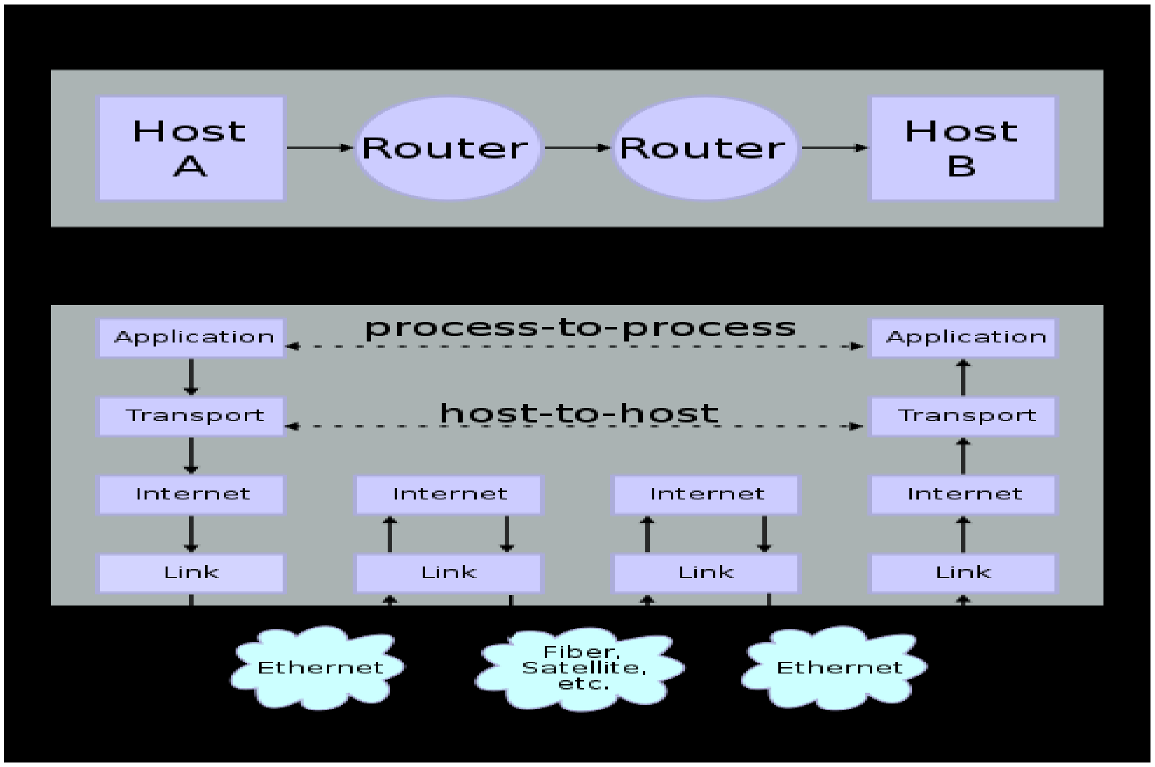


Fig: Layers and Protocols: NetWork Model



- IP
  - raw packets
  - the “Internet Layer”

- TCP
  - data stream
  - reliable, ordered
  - the "Transport Layer"
- UDP
  - user datagrams (packets)
  - unreliable, unordered
  - the "Transport Layer"

Hyper Text Transfer Protocol: To transfer the web pages (.html files) from one computer to another computer on internet.

File Transfer Protocol: To download or upload files from and to the server.

Simple Mail Transfer Protocol: To send mails on network.

Post Office Protocol: To Receive mails into the mail boxes.

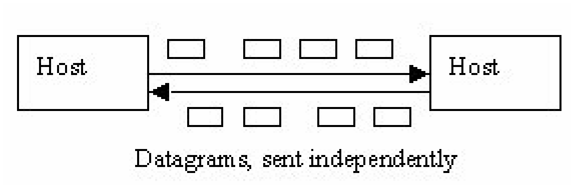
Types Of Network Programming:

Two general types are :

- Connection-oriented programming
- Connectionless Programming

Connection-oriented Networking

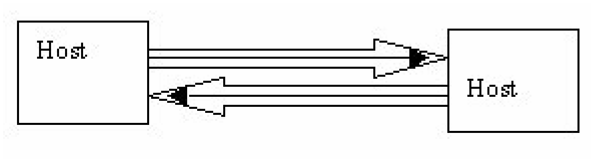
The client and server have a communication link that is open and active from the time the application is executed until it is closed. Using Internet jargon, the Transmission control protocol os a connection oriented protocol. It is reliable connection - packets are guaranteed to arrive in the order they are sent.



Connection-less Networking

The this type each instance that packets are sent, they are transmitted individually. No link to the receiver is maintained after the packets arrive. The Internet equivalent is the User Datagram Protocol (UDP). Connectionless communication is faster but not reliable.

Datagrams are used to implement a connectionless protocol, such as UDP.



URL:

Definition:

URL is an acronym for *Uniform Resource Locator* and 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.

Terminology Note:

The term *URL* can be ambiguous. It can refer to an Internet address or a `URL` object in a Java program. Where the meaning of URL needs to be specific, this text uses "URL address" to mean an Internet address and "`URL` object" to refer to an instance of the `URL` class in a program.

What Is a URL?

A URL takes the form of a string that describes how to find a resource on the Internet. URLs have two main components: the protocol needed to access the resource and the location of the resource.

Creating a URL

Within your Java programs, you can create a URL object that represents a URL address. The URL object always refers to an absolute URL but can be constructed from an absolute URL, a relative URL, or from URL components.

Parsing a URL

Gone are the days of parsing a URL to find out the host name, filename, and other information. With a valid URL object you can call any of its accessor methods to get all of that information from the URL without doing any string parsing!

Reading Directly from a URL

This section shows how your Java programs can read from a URL using the openStream() method.

Connecting to a URL

If you want to do more than just read from a URL, you can connect to it by calling openConnection() on the URL. The openConnection() method returns a URLConnection object that you can use for more general communications with the URL, such as reading from it, writing to it, or querying it for content and other information.

Reading from and Writing to a URLConnection

Some URLs, such as many that are connected to cgi-bin scripts, allow you to (or even require you to) write information to the URL. For example, a search script may require detailed query data to be written to the URL before the search can be performed. This section shows you how to write to a URL and how to get results back.

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

A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

Relative URL specifications are often used within HTML files. For example, suppose you write an HTML file called JoesHomePage.html. Within this page, are links to other pages, PicturesOfMe.html and MyKids.html, that are on the same machine and in the same directory as JoesHomePage.html. The links to PicturesOfMe.html and MyKids.html from JoesHomePage.html could be specified just as filenames, like this:

```
<a href="PicturesOfMe.html">Pictures of Me</a>
<a href="MyKids.html">Pictures of My Kids</a>
```

These URL addresses are *relative URLs*. That is, the URLs are specified relative to the file in which they are contained — JoesHomePage.html.

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.

This constructor is also useful for creating URL objects for named anchors (also called references) within a file. For example, suppose the page1.html file has a named anchor called BOTTOM at the bottom of the file. You can use the relative URL constructor to create a URL object for it like this:

```
URL page1BottomURL = new URL(page1URL,"#BOTTOM");
```

Other URL Constructors

The URL class provides two additional constructors for creating a URL object. These constructors are useful when you are working with URLs, such as HTTP URLs, that have host name, filename, port number, and reference components in the resource name portion of the URL. These two constructors are useful when you do not have a String containing the complete URL specification, but you do know various components of the URL.

For example, suppose you design a network browsing panel similar to a file browsing panel that allows users to choose the protocol, host name, port number, and filename. You can construct a URL from the panel's components. The first constructor creates a URL object from a protocol, host name, and filename. The following code snippet creates a URL to the page1.html file at the example.com site:

```
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:

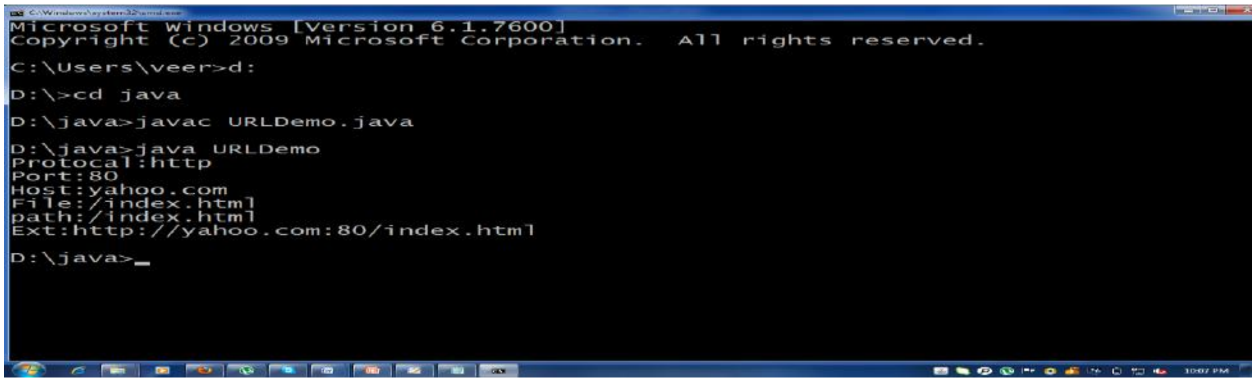
URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");

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.

```
//program to retrieve parts of url.

import  java.net.*;

public class URLDemo{

        public static void main(String args[])throws MalformedURLException{

                URL obj  = new  URL("http://yahoo.com:80/index.html");

                System.out.println("Protocal:" +  obj.getProtocol());

                System.out.println("Port:" +  obj.getPort());

                System.out.println("Host:" +  obj.getHost());

                System.out.println("File:" +  obj.getFile());

                System.out.println("path:" +  obj.getPath());

                System.out.println("Ext:" + obj.toExternalForm());

        }

}
```

OUTPUT:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>javac URLDemo.java

D:\java>java URLDemo
Protocal:http
Port:80
Host:yahoo.com
File:/index.html
path:/index.html
Ext:http://yahoo.com:80/index.html

D:\java>_
```

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
   // ...
}
```

See Exceptions for information about handling exceptions.

Note: `URL`s are "write-once" objects. Once you've created a `URL` object, you cannot change any of its attributes (protocol, host name, filename, or port number).

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`

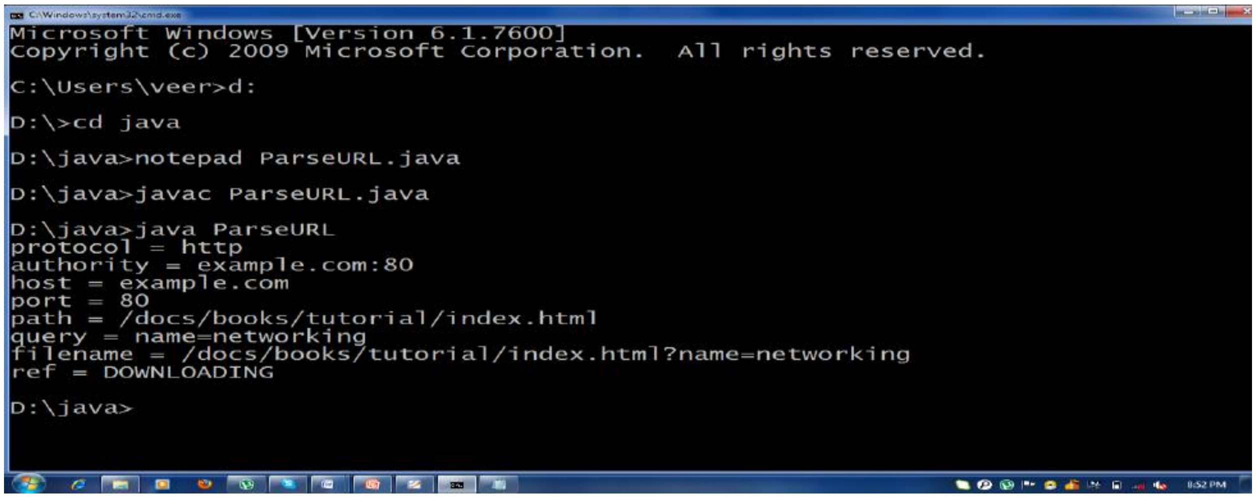
Returns the reference component of the URL.

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 get*XXX* 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());
    }
}
Output:
```



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>notepad ParseURL.java

D:\java>javac ParseURL.java

D:\java>java ParseURL
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

D:\java>
```

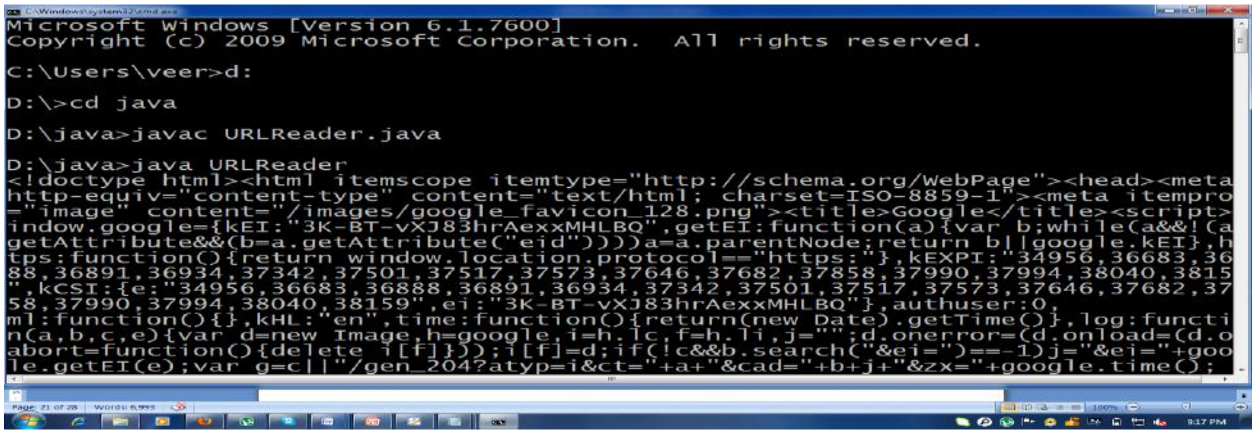
Reading Directly from a URL

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.oracle.com/. 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.oracle.com/");
       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.oracle.com/. Alternatively, the program might hang or you might see an exception stack trace. If either of the latter two events occurs, you may have to set the proxy host so that the program can find the Oracle server.



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>javac URLReader.java

D:\java>java URLReader
<!doctype html><html itemscope itemtype="http://schema.org/WebPage"><head><meta
http-equiv="content-type" content="text/html; charset=ISO-8859-1"><meta itempro
="image" content="/images/google_favicon_128.png"><title>Google</title><script>
indow.google={kEI:"3K-BT-vXJ83hrAexxMHLBQ",getEI:function(a){var b;while(a&&!(a
getAttribute&&(b=a.getAttribute("eid")))a=a.parentNode;return b||google.kEI},h
tps:function(){return window.location.protocol=="https:"},kEXPI:"34956,36683,36
88,36891,36934,37342,37501,37517,37573,37646,37682,37858,37990,37994,38040,3815
",kCSI:{e:"34956,36683,36888,36891,36934,37342,37501,37517,37573,37646,37682,37
58,37990,37994,38040,38159",ei:"3K-BT-vXJ83hrAexxMHLBQ"},authuser:0,
ml:function(){},kHL:"en",time:function(){return(new Date).getTime()},log:functi
n(a,b,c,e){var d=new Image,h=google,i=h.lc,f=h.li,j="";d.onerror=(d.onload=(d.o
abort=function(){delete i[f]});i[f]=d;if(!c&&b.search("&ei=")==-1)j="&ei="+goo
le.getEI(e);var g=c||"/gen_204?atyp=i&ct="+a+"&cad="+b+j+"&zx="+google.time();
```

Connecting to a URL

For a Java program to interact with a server-side process it simply must be able to write to a URL, thus providing data to the server. It can do this by following these steps:

1. Create a URL.
2. Retrieve the URLConnection object.
3. Set output capability on the URLConnection.
4. Open a connection to the resource.
5. Get an output stream from the connection.
6. Write to the output stream.
7. Close the output stream.

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.

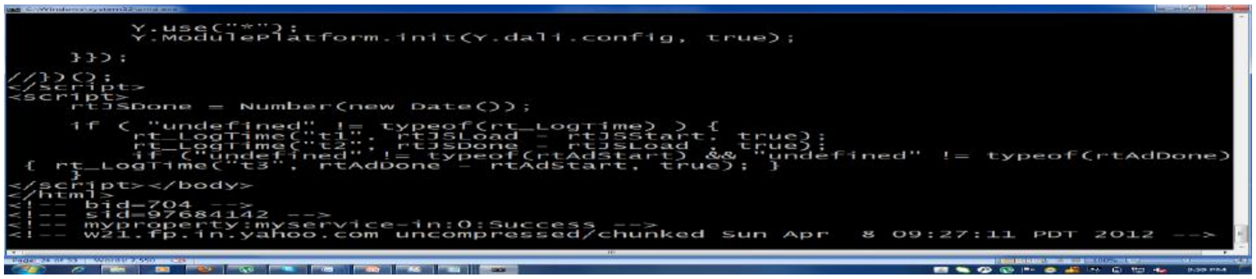
You are not always required to explicitly call the connect method to initiate the connection. Operations that depend on being connected, like getInputStream, getOutputStream, etc, will implicitly perform the connection, if necessary.

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 section shows you how.

```
// Demonstrate URLConnection.

import java.net.*;

import java.io.*;

import java.util.Date;

class UCDemo
```

```
{
public static void main(String args[]) throws Exception {
int c;
URL hp = new URL("http://www.yahoo.com");
URLConnection hpCon = hp.openConnection();
// get date
long d = hpCon.getDate();
if(d==0)
System.out.println("No date information.");
else
System.out.println("Date: " + new Date(d));
// get content type
System.out.println("Content-Type: " + hpCon.getContentType());
// get expiration date
d = hpCon.getExpiration();
if(d==0)
System.out.println("No expiration information.");
else
System.out.println("Expires: " + new Date(d));
// get last-modified date
d = hpCon.getLastModified();
if(d==0)
System.out.println("No last-modified information.");
else
System.out.println("Last-Modified: " + new Date(d));
// get content length
int len = hpCon.getContentLength();
if(len == -1)
System.out.println("Content length unavailable.");
else
System.out.println("Content-Length: " + len);
if(len != 0) {
System.out.println("=== Content ===");
InputStream input = hpCon.getInputStream();
int i = len;
while (((c = input.read()) != -1)) { // && (--i > 0)) {
System.out.print((char) c);
}
input.close();
} else {
System.out.println("No content available.");
}
}
}
```

Output:



```
        Y.use("*");
        Y.ModulePlatform.init(Y.dali.config, true);

    }});

//})();
</script>
<script>
    rtJSDone = Number(new Date());

    if ( "undefined" != typeof(rt_LogTime) ) {
        rt_LogTime("t1", rtJSLoad - rtJSStart, true);
        rt_LogTime("t2", rtJSDone - rtJSLoad , true);
        if ("undefined" != typeof(rtAdStart) && "undefined" != typeof(rtAdDone)
 { rt_LogTime("t3", rtAdDone - rtAdStart, true); }
    }
</script></body>
</html>
<!-- bid=704 -->
<!-- sid=97684142 -->
<!-- myproperty:myservice-in:0:Success -->
<!-- w21.fp.in.yahoo.com uncompressed/chunked Sun Apr  8 09:27:11 PDT 2012 -->
```

URLReadFile

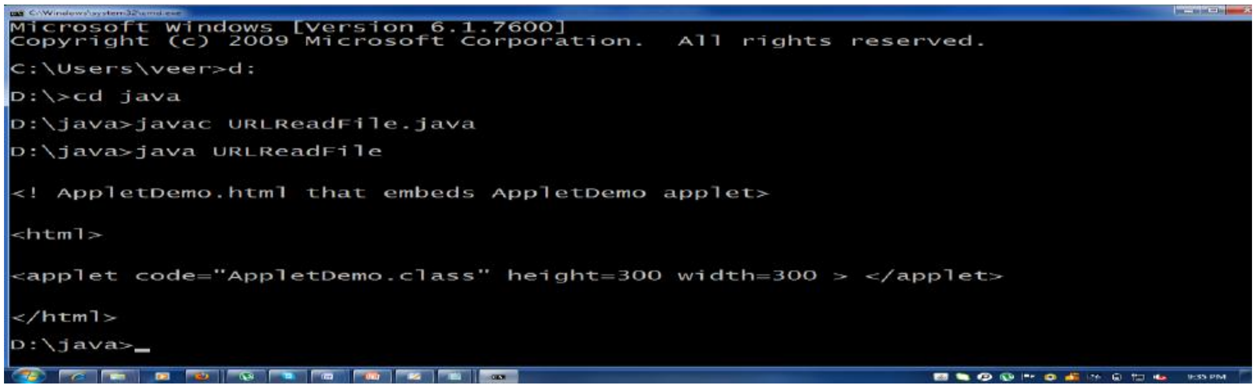
This is a simple program of java network. In this section, you will learn how to read HTML file. Here, we create a URL object and pass a url. After that, we call the URL?s **openStream()** method to get a stream from which the contents of the URL can be read. The **openStream()** method returns a **java.io.InputStream**, so reading from a URL is as easy as reading from an input stream. The simple java program uses **openStream()** method to get an input stream on the URL " file:///D:/java/AppletDemo.html" . This URL opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL. When you run the program then the HTML code and textual content from the HTML file will be displayed on the command line.

Here is the code of this program:

```
import java.net.*;

import java.io.*;

public class URLReadFile{

        public static void main(String[] args) throws IOException{

                try{

                        URL url = new URL("file:///D:/java/AppletDemo.html");

                        BufferedReader buff = new BufferedReader(new
InputStreamReader(url.openStream()));

                        String str;

                        while((str = buff.readLine()) != null) {

                                                        System.out.println("\n");

                                                        System.out.println(str);
```

```
                    }
                    buff.close();
          }
          catch(IOException e){
                    System.err.println(e);
          }
     }
}
```

Output:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.
C:\Users\veer>d:
D:\>cd java
D:\java>javac URLReadFile.java
D:\java>java URLReadFile

<! AppletDemo.html that embeds AppletDemo applet>

<html>

<applet code="AppletDemo.class" height=300 width=300 > </applet>

</html>
D:\java>_
```

ADDRESS:

Addresses are used throughout the java.net APIs as either host identifiers, or socket endpoint identifier. Address (IPv4, Ipv6) is being used as a identifier for network adapter in a network. Although two different classes exist in this package for IPv4 and IPv6 (INet4Address and INet6Address), a suggestion is to use base class InetAddress because Java will automatically adopt and use the right class so a programmer does not have to worry about protocol version.

IP addresses are 32-bit numbers. They may be written in decimal, hexadecimal, or other formats, but the most common format is dotted decimal notation. This format breaks the 32-bit address up into four bytes and writes each byte of the address as unsigned decimal integers separated by dots. For example, one of my IP addresses is 0xccD499C1. Because 0xcc = 204, 0xD4 = 212, 0x99 = 153, and 0xC1 = 193, my address in dotted decimal form is 204.212.153.193.

IP addresses are not easy to remember, even using dotted decimal notation. The Internet has adopted a mechanism, referred to as the *Domain Name System* (DNS), whereby computer names can be associated with IP addresses. These computer names are referred to as *domain names*.

The DNS has several rules that determine how domain names are constructed and how they relate to one another. For the purposes of this chapter, it is sufficient to know that domain names are computer names and that they are mapped to IP addresses.

The mapping of domain names to IP addresses is maintained by a system of *domain name servers*. These servers are able to look up the IP address corresponding to a domain name. They also provide the capability to look up the domain name associated with a particular IP address, if one exists.

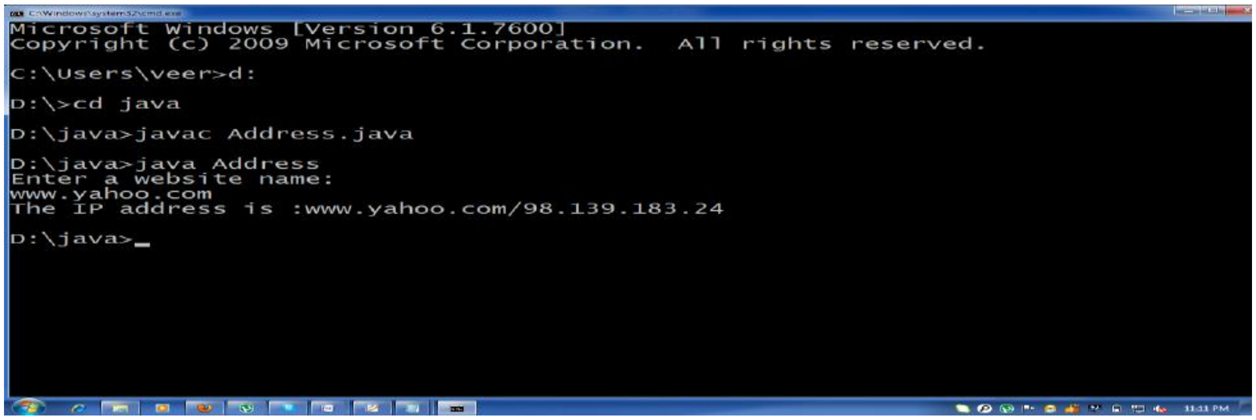
As I mentioned, IP enables communication between computers on the Internet by routing data from a source computer to a destination computer. However, computer-to-computer communication only solves half of the network communication problem. In order for an application program, such as a mail program, to communicate with another application, such as a mail server, there needs to be a way to send data to specific programs within a computer.

Find Your Host Name/IP Address:

```
//knowing IP address of a website

import java.io.*;

import java.net.*;

class Address

{

    public static void main(String args[])throws IOException

    {

        //accept name of website from keyboard

        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter a website name:");

        String site=br.readLine();

        try

        {

            //getByName() method accepts site name and returns its IP address

            InetAddress ip=InetAddress.getByName(site);

            System.out.println("The IP address is :"+ ip);
```

```
                    }
                    catch(UnknownHostException ue)
                    {
                              System.out.println("website not found");
                    }
          }
}
```

Output:

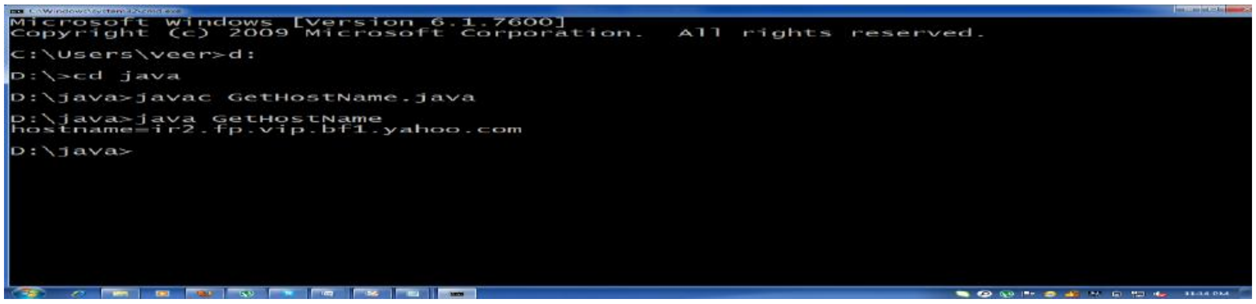


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>javac Address.java

D:\java>java Address
Enter a website name:
www.yahoo.com
The IP address is :www.yahoo.com/98.139.183.24

D:\java>_
```

program to GetHostName by entering the known IP Address

```
import java.net.*;

import java.io.*;


public class GetHostName{

 public static void main(String [] args) {

   try {

       InetAddress addr = InetAddress.getByName("98.139.183.24");

      String hostname = addr.getHostName();
      System.out.println("hostname="+hostname);

    } catch (UnknownHostException e) {

    }



    }

}
```

OUTPUT:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>javac GetHostName.java

D:\java>java GetHostName
hostname=ir2.fp.vip.bf1.yahoo.com

D:\java>
```

java.net
Class InetAddress
java.lang.Object
  └ **java.net.InetAddress**
All Implemented Interfaces:

Serializable

Direct Known Subclasses:

Inet4Address, Inet6Address

---

public class **InetAddress**
extends Object
implements Serializable

This class represents an Internet Protocol (IP) address.

An IP address is either a 32-bit or 128-bit unsigned number used by IP, a lower-level protocol on which protocols like UDP and TCP are built. The IP address architecture is defined by *RFC 790: Assigned Numbers*, *RFC 1918: Address Allocation for Private Internets*, *RFC 2365: Administratively Scoped IP Multicast*, and *RFC 2373: IP Version 6 Addressing Architecture*. An instance of an InetAddress consists of an IP address and possibly its corresponding host name (depending on whether it is constructed with a host name or whether it has already done reverse host name resolution).

Address types

Unicast	An identifier for a single interface. A packet sent to a unicast address is delivered to the interface identified by that address. The Unspecified Address -- Also called anylocal or wildcard address. It must never be assigned to any node. It indicates the absence of an address. One example of its use is as the target of bind, which allows a server to accept a client connection on any interface, in case the server host has multiple interfaces. The *unspecified* address must not be used as the destination address of an IP packet. The *Loopback* Addresses -- This is the address assigned to the loopback interface. Anything sent to this IP address loops around and becomes IP input on the local host. This address is often used when testing a client.
multicast	An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.

IP address scope

Link-local addresses are designed to be used for addressing on a single link for purposes such as auto-address configuration, neighbor discovery, or when no routers are present.

Site-local addresses are designed to be used for addressing inside of a site without the need for a global prefix.

Global addresses are unique across the internet.

Textual representation of IP addresses

The textual representation of an IP address is address family specific.

For IPv4 address format, please refer to Inet4Address#format; For IPv6 address format, please refer to Inet6Address#format.

Host Name Resolution

Host name-to-IP address *resolution* is accomplished through the use of a combination of local machine configuration information and network naming services such as the Domain Name System (DNS) and Network Information Service(NIS). The particular naming services(s) being used is by default the local machine configured one. For any host name, its corresponding IP address is returned.

Reverse name resolution means that for any IP address, the host associated with the IP address is returned.

The InetAddress class provides methods to resolve host names to their IP addresses and vise versa.

InetAddress Caching

The InetAddress class has a cache to store successful as well as unsuccessful host name resolutions. The positive caching is there to guard against DNS spoofing attacks; while the negative caching is used to improve performance.

By default, the result of positive host name resolutions are cached forever, because there is no general rule to decide when it is safe to remove cache entries. The result of unsuccessful host name resolution is cached for a very short period of time (10 seconds) to improve performance.

Under certain circumstances where it can be determined that DNS spoofing attacks are not possible, a Java security property can be set to a different Time-to-live (TTL) value for positive caching. Likewise, a system admin can configure a different negative caching TTL value when needed.

Two Java security properties control the TTL values used for positive and negative host name resolution caching:

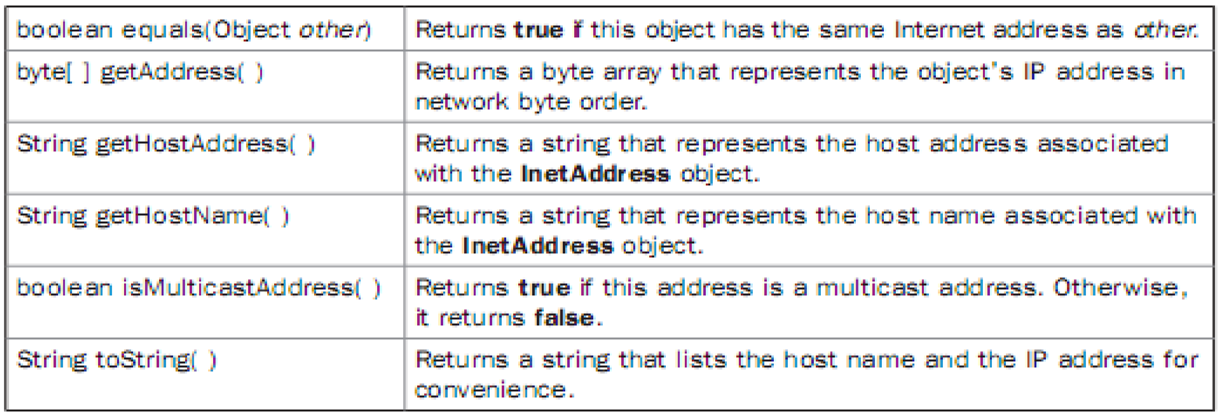
networkaddress.cache.ttl (default: -1)

> Indicates the caching policy for successful name lookups from the name service. The value is specified as as integer to indicate the number of seconds to cache the successful lookup.
>
> A value of -1 indicates "cache forever".

networkaddress.cache.negative.ttl (default: 10)

> Indicates the caching policy for un-successful name lookups from the name service. The value is specified as as integer to indicate the number of seconds to cache the failure for un-successful lookups.
>
> A value of 0 indicates "never cache". A value of -1 indicates "cache forever".

The InetAddress class encapsulates Internet addresses. It supports both numeric IP addresses and hostnames.



boolean equals(Object *other*)	Returns **true** if this object has the same Internet address as *other*.
byte[ ] getAddress( )	Returns a byte array that represents the object's IP address in network byte order.
String getHostAddress( )	Returns a string that represents the host address associated with the **InetAddress** object.
String getHostName( )	Returns a string that represents the host name associated with the **InetAddress** object.
boolean isMulticastAddress( )	Returns **true** if this address is a multicast address. Otherwise, it returns **false**.
String toString( )	Returns a string that lists the host name and the IP address for convenience.

The InetAddress class has no public variables or constructors. It provides eight access methods that support common operations on Internet addresses. Three of these methods are static.

The NSLookupApp program illustrates the use of the InetAddress class. It takes a hostname as a parameter and identifies the primary IP address associated with that host.

The InetAddress class is the abstraction representing an IP (Internet Protocol) address, it has two subclasses:

- Inet4Address for IPv4 addresses.
- Inet6Address for IPv6 addresses.

But, in most cases, there is no need to deal directly with the subclasses, as the InetAddress abstraction should cover most of needed functionalities.

Inet4Address for IPv4 addresses:

Textual representation of IPv4 address used as input to methods takes one of the following forms:

d.d.d.d

d.d.d

d.d

D

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an IPv4 address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B net- work addresses as 128.net.host.

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as net.host.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

For methods that return a textual representation as output value, the first form, i.e. a dotted-quad string, is used.

The Scope of a Multicast Address
Historically the IPv4 TTL field in the IP header has doubled as a multicast scope field: a TTL of 0 means node-local, 1 means link-local, up through 32 means site-local, up through 64 means region-local, up through 128 means continent-local, and up through 255 are global. However, the administrative scoping is preferred

About IPv6

Not all systems do have support for the IPv6 protocol, and while the Java networking stack will attempt to detect it and use it transparently when available, it is also possible to disable its use with a system property. In the case where IPv6 is not available, or explicitly disabled, Inet6Address are not valid arguments for most networking operations any more. While methods like java.net.InetAddress.getByName is guaranteed not to return an Inet6Address when looking up host names, it is possible, by passing literals, to create such an object. In which case, most methods, when called with an Inet6Address will throw an Exception.

```
// Demonstrate InetAddress.

import java.net.*;

class InetAddressTest

{

public static void main(String args[]) throws UnknownHostException {

InetAddress Address = InetAddress.getLocalHost();

System.out.println(Address);

Address = InetAddress.getByName("www.google.com");

System.out.println(Address);
```

InetAddress SW[] = InetAddress.getAllByName("www.google.com");

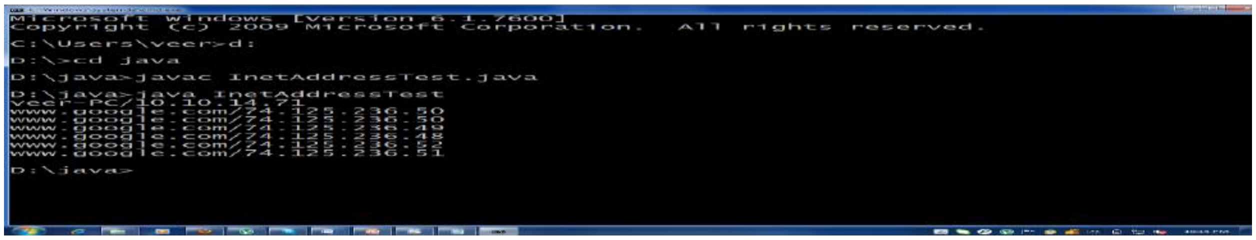
for (int i=0; i<SW.length; i++)

System.out.println(SW[i]);

}

}

OUTPUT:



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>javac InetAddressTest.java

D:\java>java InetAddressTest
veer-PC/10.10.14.71
www.google.com/74.125.236.50
www.google.com/74.125.236.50
www.google.com/74.125.236.49
www.google.com/74.125.236.48
www.google.com/74.125.236.52
www.google.com/74.125.236.51

D:\java>
```

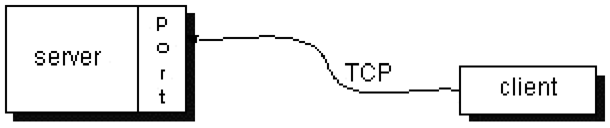
PORT:

Understanding Ports

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.

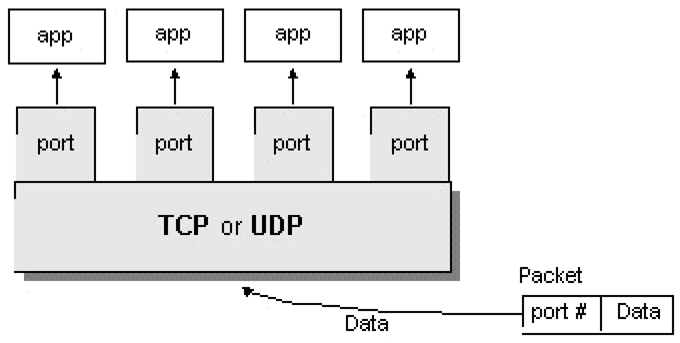
In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:



Definition:

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

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, as illustrated in this figure:

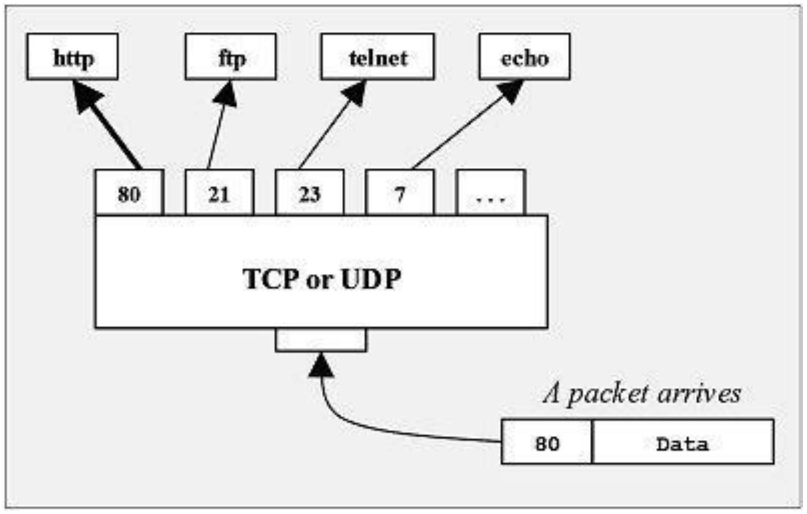


Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*. Your applications should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

While the IP address will get the packet to the right computer, how does the packet get to the right program? A computer might have:

- Several programs running at the same time
- Several programs trying to communicate via the Internet
- All the programs communicating out the same physical ethernet cable

The *Port* number is the key for organizing all of this. The packets are guided to the correct program by the operating system according to the port number as indicated in this diagram:.



The port is a 16-bit number. It is occasionally seen on Web addresses:

http://www.someplace.com:80/

where 80 is the default for the HTTP server. Nowadays the port number is seldom included in the URL since the default is almost always used.

Various applications using particular protocols use standard port values as shown in tthe above diagram. Unix machines reserve ports 1-1023 for standard services. Windows machines do not restrict the use of these ports but to make your Java programs portable, it is wise to use port values above 1023.

One type of firewall assigns port numbers to the machines behind its shield. Incoming packets all go to the same IP address but with different port numbers according to the machine they are destined for. This both reduces the exposure and accessibility to the machines and reduces the need for universally unique IP numbers. Behind the firewall, the local addresses can be the same as in other LANs since the LANs are isolated from each other.

This is one of the reasons that four bytes remain sufficient for the internet despite the explosion in the number of devices with IP addresses. (Nevertheless, a sixteen byte version of IP addressing called IPv6 will gradually take over.)

- Port: a meeting place on a host
    1. one service per port
    2. 1-1023 = well-known services
    3. 1024+ = experimental services, temporary

❖ Well-Known Ports

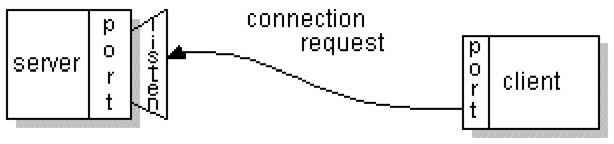
- 20,21: FTP
- 23: telnet
- 25: SMTP

- 43: whois
- 80: HTTP
- 119: NNTP
- 1099: RMI

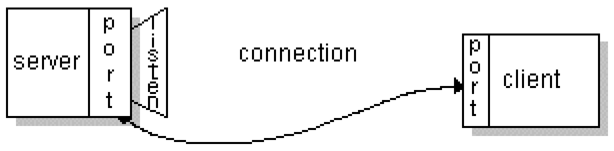
SOCKET:

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. This lesson shows you how to use the `Socket` and `ServerSocket` classes.

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. See Working with URLs for information about connecting to the Web via URLs.

Class Socket

```
java.lang.Object
  └ java.net.Socket
```

Direct Known Subclasses:

SSLSocket

```
public class Socket
extends Object
```

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

The actual work of the socket is performed by an instance of the `SocketImpl` class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.

Constructor Summary	
	Socket() Creates an unconnected socket, with the system-default type of SocketImpl.
	Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the

	specified IP address.
	Socket(InetAddress host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
	Socket(Proxy proxy) Creates an unconnected socket, specifying the type of proxy, if any, that should be used regardless of any other settings.
protected	**Socket**(SocketImpl impl) Creates an unconnected Socket with a user-specified SocketImpl.
	Socket(String host, int port) Creates a stream socket and connects it to the specified port number on the named host.
	Socket(String host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(String host, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote host on the specified remote port.

Method Summary	
void	**bind**(SocketAddress bindpoint) Binds the socket to a local address.
void	**close**() Closes this socket.
void	**connect**(SocketAddress endpoint) Connects this socket to the server.
void	**connect**(SocketAddress endpoint, int timeout)

	Connects this socket to the server with a specified timeout value.
SocketChannel	**getChannel**() Returns the unique SocketChannel object associated with this socket, if any.
InetAddress	**getInetAddress**() Returns the address to which the socket is connected.
InputStream	**getInputStream**() Returns an input stream for this socket.
boolean	**getKeepAlive**() Tests if SO_KEEPALIVE is enabled.
InetAddress	**getLocalAddress**() Gets the local address to which the socket is bound.
int	**getLocalPort**() Returns the local port to which this socket is bound.
SocketAddress	**getLocalSocketAddress**() Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.
boolean	**getOOBInline**() Tests if OOBINLINE is enabled.
OutputStream	**getOutputStream**() Returns an output stream for this socket.
int	**getPort**() Returns the remote port to which this socket is connected.
int	**getReceiveBufferSize**() Gets the value of the SO_RCVBUF option for this Socket, that is the buffer size used by the platform for input on this Socket.
SocketAddress	**getRemoteSocketAddress**() Returns the address of the endpoint this socket is connected to, or null if it is unconnected.
boolean	**getReuseAddress**()

	Tests if SO_REUSEADDR is enabled.
int	**getSendBufferSize**() Get value of the SO_SNDBUF option for this Socket, that is the buffer size used by the platform for output on this Socket.
int	**getSoLinger**() Returns setting for SO_LINGER.
int	**getSoTimeout**() Returns setting for SO_TIMEOUT.
boolean	**getTcpNoDelay**() Tests if TCP_NODELAY is enabled.
int	**getTrafficClass**() Gets traffic class or type-of-service in the IP header for packets sent from this Socket
boolean	**isBound**() Returns the binding state of the socket.
boolean	**isClosed**() Returns the closed state of the socket.
boolean	**isConnected**() Returns the connection state of the socket.
boolean	**isInputShutdown**() Returns whether the read-half of the socket connection is closed.
boolean	**isOutputShutdown**() Returns whether the write-half of the socket connection is closed.
void	**sendUrgentData**(int data) Send one byte of urgent data on the socket.
void	**setKeepAlive**(boolean on) Enable/disable SO_KEEPALIVE.
void	**setOOBInline**(boolean on) Enable/disable OOBINLINE (receipt of TCP urgent data) By default, this option is disabled and TCP urgent data received on a socket is silently

	discarded.
void	**setPerformancePreferences**(int connectionTime, int latency, int bandwidth) Sets performance preferences for this socket.
void	**setReceiveBufferSize**(int size) Sets the SO_RCVBUF option to the specified value for this Socket.
void	**setReuseAddress**(boolean on) Enable/disable the SO_REUSEADDR socket option.
void	**setSendBufferSize**(int size) Sets the SO_SNDBUF option to the specified value for this Socket.
static void	**setSocketImplFactory**(SocketImplFactory fac) Sets the client socket implementation factory for the application.
void	**setSoLinger**(boolean on, int linger) Enable/disable SO_LINGER with the specified linger time in seconds.
void	**setSoTimeout**(int timeout) Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
void	**setTcpNoDelay**(boolean on) Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).
void	**setTrafficClass**(int tc) Sets traffic class or type-of-service octet in the IP header for packets sent from this Socket.
void	**shutdownInput**() Places the input stream for this socket at "end of stream".
void	**shutdownOutput**() Disables the output stream for this socket.
String	**toString**() Converts this socket to a String.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Class SocketAddress

java.lang.Object
  └ **java.net.SocketAddress**
All Implemented Interfaces:
    Serializable
Direct Known Subclasses:
    InetSocketAddress

public abstract class **SocketAddress**
extends Object
implements Serializable

This class represents a Socket Address with no protocol attachment. As an abstract class, it is meant to be subclassed with a specific, protocol dependent, implementation.

- A *network socket* is a lot like an electrical socket.
- Socket: a two-way connection
- *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.
- *Transmission Control Protocol (TCP)* is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit your data.
- A third protocol, *User Datagram Protocol (UDP)*, sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

The Socket Class(Client)

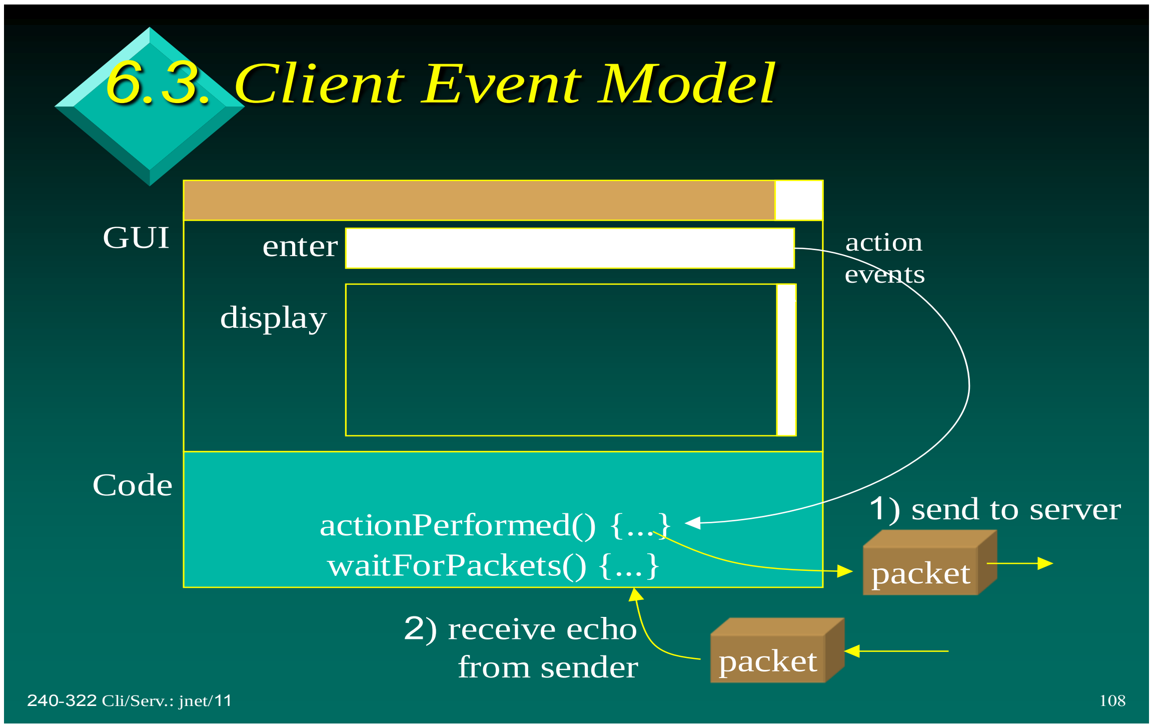
Socket s = new Socket(“www.starwave.com”, 90);

- Socket(String host, int port)
- InputStream getInputStream()
- OutputStream getOutputStream()
- void close()

This client program is straightforward and simple because the Echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.



The java.net.ServerSocket Class

- The java.net.ServerSocket class represents a server socket.
- A ServerSocket object is constructed on a particular local port. Then it calls accept() to listen for incoming connections.
- accept() blocks until a connection is detected. Then accept() returns a java.net.Socket object that performs the actual communication with the client.

Constructors

- There are three constructors that specify the port to bind to, the queue length for incoming connections, and the IP address to bind to:

public ServerSocket(int port) throws IOException

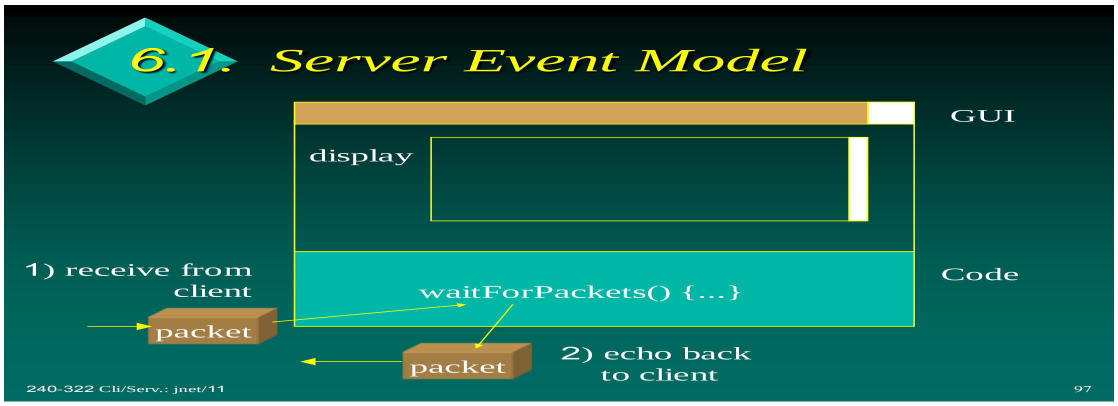
public ServerSocket(int port, int backlog) throws
                        IOException

public ServerSocket(int port, int backlog, InetAddress networkInterface) throws IOException

Constructing Server Sockets

- specify the port number to listen :

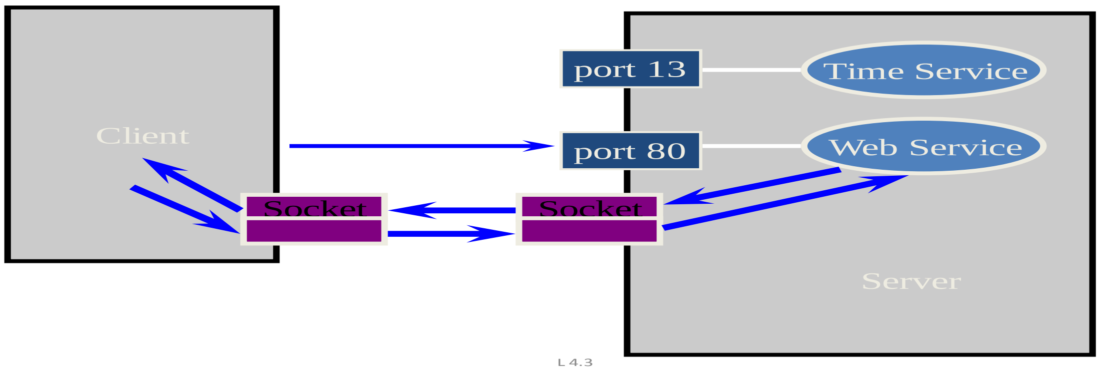
```
try {
  ServerSocket ss = new ServerSocket(80);
}
catch (IOException e) {
  System.err.println(e);
}
```



Simple Client Server Program:

Socket Programming: is a low level way to establish connection between server and a client with the help of a Socket.

Sockets and Ports



· Data will be sent or received through the Socket.

· Socket at server side is called Server Socket.

· Socket at client side is called Client Socket.

· The id number alloted to a Socket is called port number.

· When a new socket is created a new port number is alloted.

· When the service on the socket is changed we should change the port number.

· A Socket is a point of connection between a server and client.

· Server Socket is created using ServerSocket class.

· Client Socket is created using Socket class.

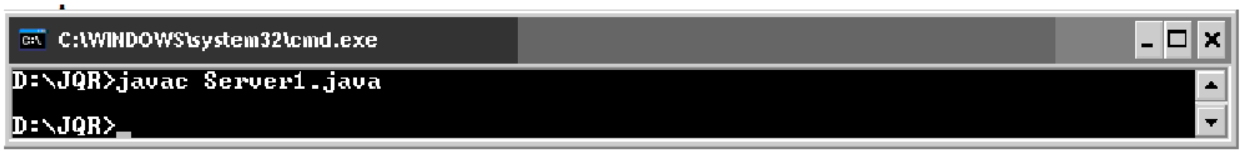
· ServerSocket and Socket classes are available in java.net package.

- Client - initiates connection
  - retrieves data,
  - displays data,
  - responds to user input,
  - requests more data
- Examples:
  - Web Browser
  - Chat Program
  - PC accessing files

Program : Write a program to create a server for sending some strings to the client.

```
//A server that sends message to client

import java.net.*;

import java.io.*;

class Server1

{  public static void main(String args[]) throws IOException

  {  //Create Server side socket

    ServerSocket ss = new ServerSocket (777);

    //make this socket accept client connection

    Socket s = ss.accept ();

    System.out.println ("A connection established...");

    //attach OutputStream to socket

    OutputStream obj = s.getOutputStream ();

    //to send data to Socket

    PrintStream ps = new PrintStream (obj);

    //now send the data

    String str = "Hello Client";

    ps.println (str);

    ps.println ("Bye");

    //close connection

    s.close ();

    ss.close ();

    ps.close ();
  }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Server1.java

D:\JQR>_
```

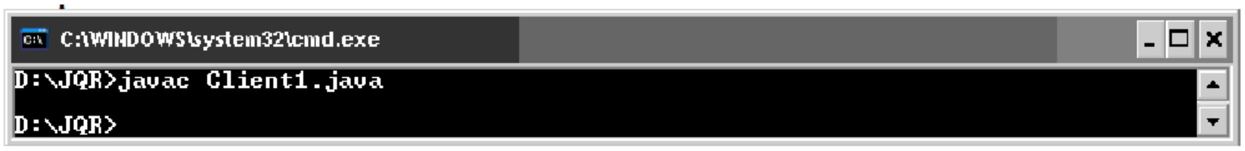
Note: Do not run this program till client is also created.

Program : Write a program to create a client which accepts all the strings sent by the server

```
// a client that accepts data from server
import java.util.*;
import java.io.*;
import java.net.*;
class Client1
{  public static void main(String args[]) throws IOException
  {  //Create client socket
    Socket s = new Socket ("localhost", 777);
    //attach input stream to Socket
    InputStream obj = s.getInputStream ();
    //to receive data from socket
    BufferedReader br = new BufferedReader (new InputStreamReader (obj));
    //read data coming from server
    String str;
    while ((str = br.readLine() ) != null )
```

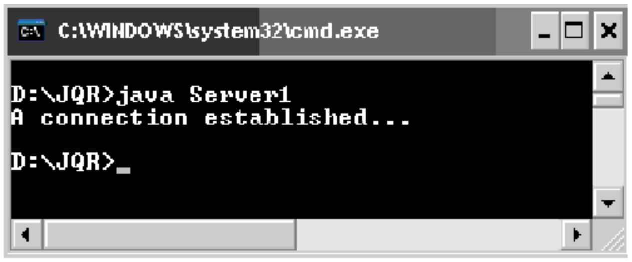
```
      System.out.println (str);

    //close connection

    s.close ();

    br.close ();

  }

}
```

Output:

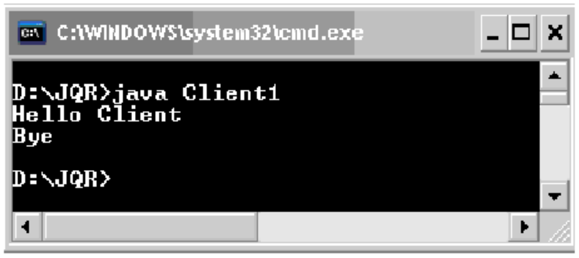


```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Client1.java

D:\JQR>
```

After compiling Server1.java and Client1.java, run these programs in two separate dos windows:



```
C:\WINDOWS\system32\cmd.exe

D:\JQR>java Server1
A connection established...

D:\JQR>_
```

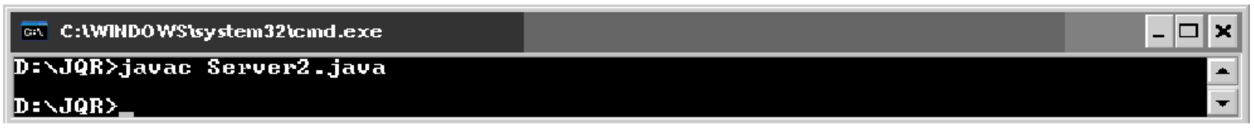


```
C:\WINDOWS\system32\cmd.exe

D:\JQR>java Client1
Hello Client
Bye

D:\JQR>
```

Program : Write a program to create a server such that the server receives data from the client using BufferedReader and then sends reply to the client using PrintStream.

```
//chat server

import java.net.*;

import java.io.*;

class Server2

{  public static void main(String args[]) throws IOException

  {  //create a server socket

    ServerSocket ss = new ServerSocket (888);
    //accept client connection
    Socket s = ss.accept ();
    System.out.println ("A connection is established...");
    //to send data to client
    PrintStream ps = new PrintStream (s.getOutputStream ());
    //to receive data from client
    BufferedReader br = new BufferedReader (new InputStreamReader
                                                        (s.getInputStream ()));
    //to accept data from keyboard to sent to client
    BufferedReader kb = new BufferedReader (new InputStreamReader (System.in));
while (true)   //server runs continuously
{  String str, str1;
  while ( (str= br.readLine() ) != null)
  {  System.out.println (str);
    str1 = kb.readLine ();
    ps.println (str1);
  }
  s.close ();  ss.close ();
  System.exit (0);
    }
  }
}
```

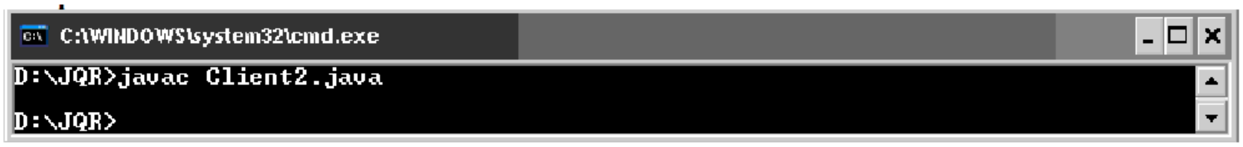
Output:


```
D:\JQR>javac Server2.java

D:\JQR>_
```

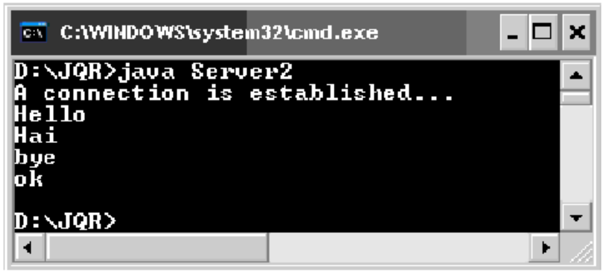
Program : Write a program to create a client which first connects to a server then starts the communication by sending a string to the server.

```
//chat client
import java.net.*;
import java.io.*;
class Client2
{  public static void main(String args[]) throws IOException
  {  //create a client socket
    Socket s = new Socket ("localhost", 888);
    //to send data to server
    DataOutputStream dos = new DataOutputStream (s.getOutputStream ());
    //to receive data from server
    BufferedReader br = new BufferedReader (new InputStreamReader
  (s.getInputStream ()));
    //to read data from keyboard to send to server
    BufferedReader kb = new BufferedReader (new InputStreamReader (System.in));
    //now communicate with Server
    String str, str1;
    while ( !(str = kb.readLine() ).equals("exit") )
    {  dos.writeBytes (str+ "\n");
      str1 = br.readLine ();
      System.out.println (str1);
    }
    s.close ();
  }
}
```

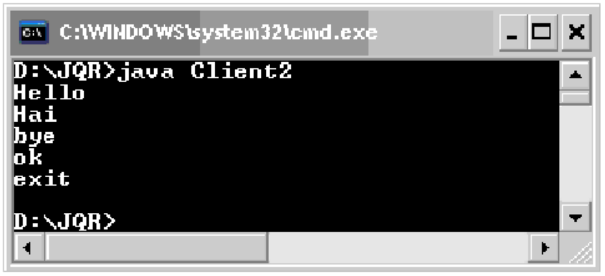
Output:



```
D:\JQR>javac Client2.java

D:\JQR>
```



```
D:\JQR>java Server2
A connection is established...
Hello
Hai
bye
ok

D:\JQR>
```



```
D:\JQR>java Client2
Hello
Hai
bye
ok
exit

D:\JQR>
```

EXTRA INFO: A Simple Server and Client Examples

A simple client and the server programs are explained for all the types of techniques. A code samples for the connection-oriented, next for the connectionless and then for broadcasting are as follows.

Connection-oriented Client and Server : (TCP)

The client sends the message *" Hi from client "* and the server response will be *" Hello from Server "*.

Server Program

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

public class Server{

//port number should be more than 1024

public static final int PORT = 1025;

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

```
ServerSocket sersock = null;
Socket sock = null;
System.out.println(" Wait !! ");

try
{
 //  Initialising the ServerSocket
 sersock =  new ServerSocket(PORT);

 // Gives the Server Details Machine name,
 Port number

 System.out.println("Server Started  :"+sersock);

 try
 {

  // makes a socket connection to particular
  client after
  // which two way communication take place

  sock = sersock.accept();

  System.out.println("Client Connected  :"+ sock);

  // Receive message from client i.e Request
  from client

  DataInputStream ins = new
  DataInputStream(sock.getInputStream());
  // Send message to the client i.e Response

  PrintStream ios = new
  sPrintStream(sock.getOutputStream());
  ios.println("Hello from server");
  ios.close();

  // Close the Socket connection

   sock.close();

   }
       catch(SocketException se)
       {
   System.out.println("Server Socket
        problem  "+se.getMessage());
   }
       catch(Exception e)
       {
   System.out.println("Couldn't start "
                 + e.getMessage()) ;
   }

// Usage of some methods in Socket class

 System.out.println(" Connection from :  " +
 sock.getInetAddress());
 } // main

}  // Server class
```

Client Program

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


class client
{
 Public static void main(String args[])
 {
 Socket sock=null;
 DataInputStream dis=null;
 PrintStream ps=null;
 System.out.println(" Trying to connect");

 try
 {
 // to get the ip address of the
 server by the name

 InetAddress ip =InetAddress.getByName
 ("Hari.calsoftlabs.com");

 // Connecting to the port 1025 declared
 in the Serverclass
 // Creates a socket with the server
  bind to it.

  sock= new Socket(ip,Server.PORT);
  ps= new PrintStream(sock.getOutputStream());
  ps.println(" Hi from client");
  DataInputStream is = new
  DataInputStream(sock.getInputStream());
  System.out.println(is.readLine());

 }
 0catch(SocketException e)
 {
  System.out.println("SocketException " + e);
 }
 catch(IOException e)
 {
  System.out.println("IOException " + e);
 }

  // Finally closing the socket from
  the client side

 finally
 {
```

```
 try
  {
   sock.close();
  }
  catch(IOException ie)
  {
   System.out.println(" Close Error   :" +
   ie.getMessage());
  }
 }  // finally

} // main
}   // Class Client
```

Running the Server and Client

After you've successfully compiled the server and the client programs, you run them. You have to run the server program first. Just use the Java interpreter and specify the Server class name. Once the server has started, you can run the client program. After the client sends a request and receives a response from the server, you should see output similar to this :

```
On client side:
   Hello from Server
On Server side:
   Hi from client
```

Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman.

Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes:

The **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the DatagramPackets. Each is examined here.

DatagramSocket:

DatagramSocket defines four public constructors. They are shown here:

- DatagramSocket( ) throws SocketException

The first creates a DatagramSocket bound to any unused port on the local computer.

- DatagramSocket(int port) throws SocketException

The second creates a DatagramSocket bound to the port specified by port.

- DatagramSocket(int port, InetAddress ipAddress) throws SocketException

The third constructs a DatagramSocket bound to the specified port and InetAddress.

- DatagramSocket(SocketAddress address) throws SocketException

The fourth constructs a DatagramSocket bound to the specified SocketAddress.

SocketAddress is an abstract class that is implemented by the concrete class InetSocketAddress. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a SocketException if an error occurs while creating the socket.

DatagramSocket defines many methods.

Two of the most important are **send( ) and receive( )**, which are shown here:

- void send(DatagramPacket packet) throws IOException

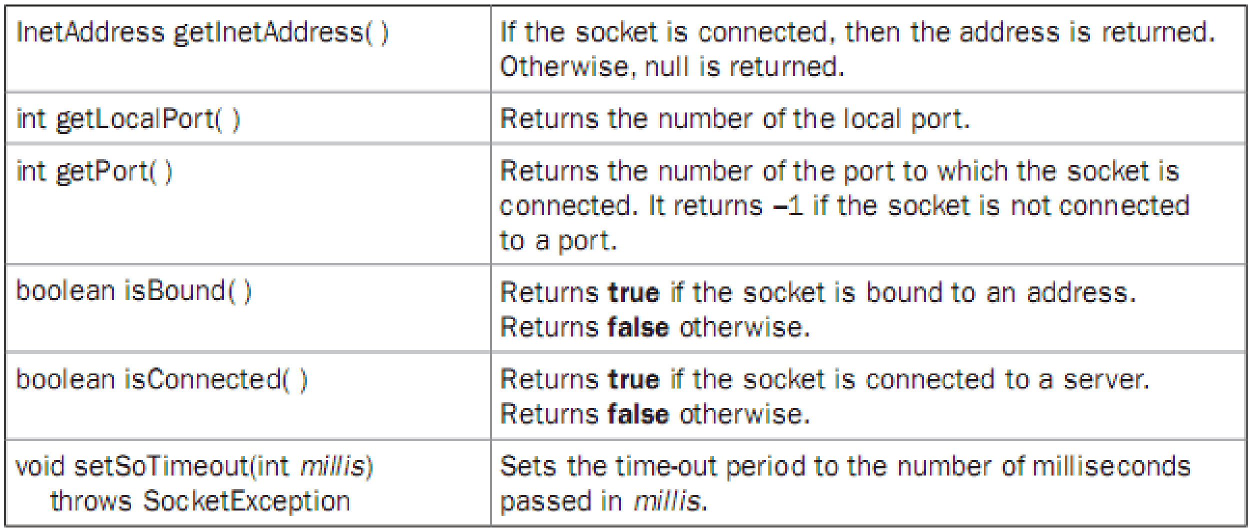
The send( ) method sends packet to the port specified by packet.

- void receive(DatagramPacket packet) throws IOException

The receive method waits for a packet to be received from the port specified by packet and returns the result.

Other methods give you access to various attributes associated with a DatagramSocket.

Here is a sampling:



InetAddress getInetAddress( )	If the socket is connected, then the address is returned. Otherwise, null is returned.
int getLocalPort( )	Returns the number of the local port.
int getPort( )	Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port.
boolean isBound( )	Returns **true** if the socket is bound to an address. Returns **false** otherwise.
boolean isConnected( )	Returns **true** if the socket is connected to a server. Returns **false** otherwise.
void setSoTimeout(int *millis*) throws SocketException	Sets the time-out period to the number of milliseconds passed in *millis*.

DatagramPacket:

DatagramPacket defines several constructors. Four are shown here:

- DatagramPacket(byte data[ ], int size)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a DatagramSocket.

- DatagramPacket(byte data[ ], int offset, int size)

The second form allows you to specify an offset into the buffer at which data will be stored.

- DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)

The third form specifies a target address and port, which are used by a DatagramSocket to determine where the data in the packet will be sent.

- DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)

The fourth form transmits packets beginning at the specified offset into the data.

Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope.

DatagramPacket defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length. In general, the get methods are used on packets that are received and the set methods are used on packets that will be sent.

InetAddress getAddress( )	Returns the address of the source (for datagrams being received) or destination (for datagrams being sent).
byte[ ] getData( )	Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.
int getLength( )	Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This may not equal the length of the whole byte array.
int getOffset( )	Returns the starting index of the data.
int getPort( )	Returns the port number.
void setAddress(InetAddress *ipAddress*)	Sets the address to which a packet will be sent. The address is specified by *ipAddress*.
void setData(byte[ ] *data*)	Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*.
void setData(byte[ ] *data*, int *idx*, int *size*)	Sets the data to *data*, the offset to *idx*, and the length to *size*.
void setLength(int *size*)	Sets the length of the packet to *size*.
void setPort(int *port*)	Sets the port to *port*.

A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

Connectionless Client and Server : (UDP)

A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

The java.net package contains two classes to help you write Java programs that use datagrams to send and receive packets over the network: DatagramSocket, DatagramPacket, and MulticastSocket An application can send and receive DatagramPackets through a DatagramSocket. In addition, DatagramPackets can be broadcast to multiple recipients all listening to a MulticastSocket.

The following source code demonstrates a slightly more complex server that uses datagrams instead of sockets. After a connection is made, It echoes back whatever is sent by the client instead of simply hanging up the connection. It is Called as echo Server.

Server Program

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


public class EchoServer
{

//Initialize Port number and Packet Size

 static final int serverPort = 1026;
 static final int packetSize = 1024;

 public static void main(String args[])
 throws SocketException{

 DatagramPacket packet;
 DatagramSocket socket;
 byte[] data;    // For data to be
 Sent in packets
 int clientPort;
 InetAddress address;
 String str;

 socket = new DatagramSocket(serverPort);

 for(;;){
 data = new byte[packetSize];

 // Create packets to receive the message

 packet = new DatagramPacket(data,packetSize);
 System.out.println("Waiting to receive
 the packets");

 try{

 // wait infinetely for arrive of the packet

 socket.receive(packet);

 }catch(IOException ie)
 {
 System.out.println(" Could not Receive
 :"+ie.getMessage());
 System.exit(0);
 }

 // get data about client in order to
 echo data back

 address = packet.getAddress();
 clientPort = packet.getPort();

 // print string that was received
 on server's console
 str = new String(data,0,0,packet.getLength());
 System.out.println("Message  :"+ str.trim());
 System.out.println("From   :"+address);

 // echo data back to the client
 // Create packets to send to the client

 packet = new DatagramPacket(data,packetSize,
 address,clientPort);

 try
 {
 // sends packet

 socket.send(packet);

 }
 catch(IOException ex)
 {
  System.out.println("Could not Send
  "+ex.getMessage());
  System.exit(0);
  }

 } // for loop

 } // main


} // class EchoServer
```

Client Program

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


public class EchoClient{

 static final int serverPort = 1026;
 static final int packetSize = 1024;

 public static void main(String args[]) throws
 UnknownHostException, SocketException{
 DatagramSocket socket; //How we send packets
 DatagramPacket packet; //what we send it in
 InetAddress address; //Where to send
 String messageSend; //Message to be send
 String messageReturn; //What we get back
 from the Server
 byte[] data;

 //Checks for the arguments that sent to
 the java interpreter
 // Make sure command line parameters correctr

 if(args.length != 2)
 {
```

```
System.out.println("Usage Error :
Java EchoClient < Server name> < Message>");
System.exit(0);
}

// Gets the IP address of the Server
address = InetAddress.getByName(args[0]);
socket = new DatagramSocket();

data = new byte[packetSize];
messageSend = new String(args[1]);
messageSend.getBytes
(0,messageSend.length(),data,0);

// remember datagrams hold bytes
packet = new
DatagramPacket(data,data.length,address,serverPort);
System.out.println(" Trying to Send the packet ");

try
{
 // sends the packet

 socket.send(packet);

 }
 catch(IOException ie)
 {
 System.out.println("Could not Send :"+ie.getMessage());
 System.exit(0);
 }

 packet is reinitialized to use it for recieving

 packet = new DatagramPacket(data,data.length);

 try
 {
 // Receives the packet from the server

 socket.receive(packet);

 }
 catch(IOException iee)
 {
 System.out.println("Could not receive :
 "+iee.getMessage() );
 System.exit(0);
 }

 // display message received

 messageReturn = new String (packet.getData(),0);
 System.out.println("Message Returned : "+
 messageReturn.trim());
 }    // main
 } // Class EchoClient
```

Running the Server and Client

The client side and the server side networking code looks actually very similar.This is true with many applications that use datagrams because the java.net.DatagramSocket class is used to both send and receive DatagramPackets.

Suppose server running on the machine named Hari.calsoftlabs.com, whereas the client running on the xxxx.info.com. As you can see at the end of the example the server is running waiting for the another connection, while the execution of the client has halted.

Server Side :

To start the server :

```
Java EchoServer
```

Output:

```
Message:Hello
From   :xxxx.info.com
```

Client Side :

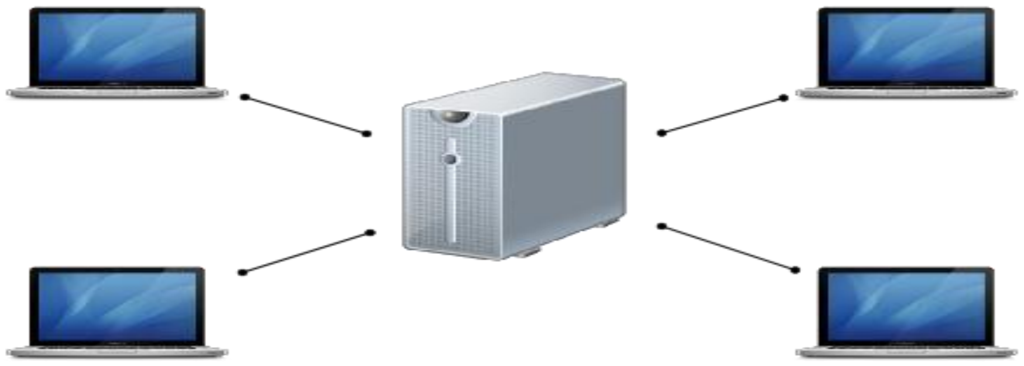
```
Java EchoClient Hari.calsoftlabs.com Hello
```

Output:

```
Message Returned : Hello
```

Multiple Clients:

- Multiple clients can connect to the same port on the server at the same time.
- Incoming data is distinguished by the port to which it is addressed and the client host and port from which it came.
- The server can tell for which service (like http or ftp) the data is intended by inspecting the port.
- It can tell which open socket on that service the data is intended for by looking at the client address and port stored with the data.



Java Programming Language Basics , finished with a simple network communications example using the Remote Method Invocation (RMI) application programming interface (API). The RMI example allows multiple client programs to communicate with the same server program without any explicit code to do this because the RMI API is built on sockets and threads.

This lesson presents a simple sockets-based program to introduce the concepts of sockets and multi-threaded programming. A multi-threaded program performs multiple tasks at one time such as fielding simultaneous requests from many client programs.

What are Sockets and Threads?

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it runs so any client program anywhere in the network with a socket associated with that same port can communicate with the server program.

A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request.

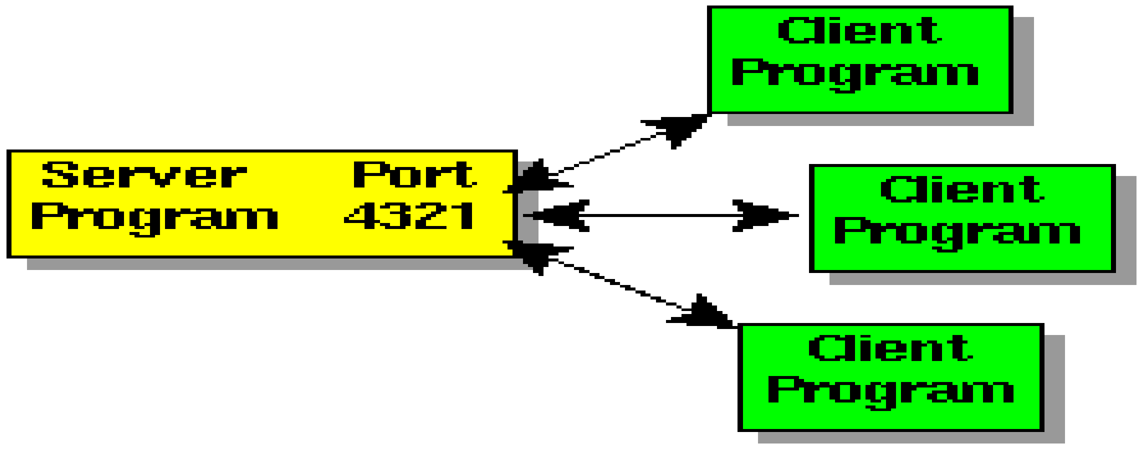


Fig: Server handles the multiple clients

One way to handle requests from more than one client is to make the server program multi-threaded. A multi-threaded server creates a thread for each communication it accepts from a client. A thread is a sequence of instructions that run independently of the program and of any other threads.

Using threads, a multi-threaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

Multithreaded Server Example

The example in its current state works between the server program and one client program only. To allow multiple client connections, the server program has to be converted to a multithreaded server program.

```
import java.awt.Color;

import java.awt.BorderLayout;

import java.awt.event.*;

import javax.swing.*;



import java.io.*;

import java.net.*;



class ClientWorker implements Runnable {
```

```
private Socket client;
private JTextArea textArea;

ClientWorker(Socket client, JTextArea textArea) {
 this.client = client;
 this.textArea = textArea;
}

public void run(){
  String line;
  BufferedReader in = null;
  PrintWriter out = null;
  try{
    in = new BufferedReader(new InputStreamReader(client.getInputStream()));
    out = new PrintWriter(client.getOutputStream(), true);
  } catch (IOException e) {
    System.out.println("in or out failed");
    System.exit(-1);
  }

  while(true){
    try{
      line = in.readLine();
//Send data back to client
         out.println(line);
         textArea.append(line);
       } catch (IOException e) {
         System.out.println("Read failed");
         System.exit(-1);
       }
    }
  }
}

class SocketThrdServer extends JFrame{

   JLabel label = new JLabel("Text received over socket:");
   JPanel panel;
   JTextArea textArea = new JTextArea();
   ServerSocket server = null;

   SocketThrdServer(){ //Begin Constructor
     panel = new JPanel();
     panel.setLayout(new BorderLayout());
     panel.setBackground(Color.white);
     getContentPane().add(panel);
     panel.add("North", label);

     panel.add("Center", textArea);

   } //End Constructor


  public void listenSocket(){

    try{

      server = new ServerSocket(4444);

    } catch (IOException e) {

      System.out.println("Could not listen on port 4444");

      System.exit(-1);

    }

    while(true){

      ClientWorker w;

      try{

        w = new ClientWorker(server.accept(), textArea);

        Thread t = new Thread(w);

        t.start();

      } catch (IOException e) {

        System.out.println("Accept failed: 4444");

        System.exit(-1);

      }

    }

  }
```

```
  protected void finalize(){
//Objects created in run method are finalized when
//program terminates and thread exits
     try{
        server.close();
    } catch (IOException e) {
        System.out.println("Could not close socket");
        System.exit(-1);
    }
  }

  public static void main(String[] args){
        SocketThrdServer frame = new SocketThrdServer();
            frame.setTitle("Server Program");
        WindowListener l = new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        System.exit(0);
                }
        };
        frame.addWindowListener(l);
        frame.pack();
        frame.setVisible(true);
        frame.listenSocket();

  }

}
```

Program Explanation:

In this example the listenSocket method loops on the server.accept call waiting for client connections and creates an instance of the ClientWorker class for each client connection it accepts. The textArea component that displays the text received from the client connection is passed to the ClientWorker instance with the accepted client connection.

```
public void listenSocket(){
  try{
    server = new ServerSocket(4444);
  } catch (IOException e) {
    System.out.println("Could not listen on port 4444");
    System.exit(-1);
  }
  while(true){
    ClientWorker w;
    try{
//server.accept returns a client connection
      w = new ClientWorker(server.accept(), textArea);
      Thread t = new Thread(w);
      t.start();
    } catch (IOException e) {
      System.out.println("Accept failed: 4444");
      System.exit(-1);
    }
  }
}
```

The important changes in this version of the server program over the non-threaded server program are the line and client variables are no longer instance variables of the server class, but are handled inside the ClientWorker class.

The ClientWorker class implements the Runnable interface, which has one method, run. The run method executes independently in each thread. If three clients request connections, three ClientWorker instances are created, a thread is started for each ClientWorker instance, and the run method executes for each thread.

In this example, the run method creates the input buffer and output writer, loops on the input stream waiting for input from the client, sends the data it receives back to the client, and sets the text in the text area.

```
class ClientWorker implements Runnable {
  private Socket client;
  private JTextArea textArea;

//Constructor
  ClientWorker(Socket client, JTextArea textArea) {
    this.client = client;
    this.textArea = textArea;
  }

  public void run(){
    String line;
    BufferedReader in = null;
    PrintWriter out = null;
    try{
      in = new BufferedReader(new
        InputStreamReader(client.getInputStream()));
      out = new
        PrintWriter(client.getOutputStream(), true);
    } catch (IOException e) {
      System.out.println("in or out failed");
      System.exit(-1);
    }

    while(true){
      try{
        line = in.readLine();
//Send data back to client
        out.println(line);
//Append data to text area
        textArea.append(line);
       }catch (IOException e) {
        System.out.println("Read failed");
        System.exit(-1);
       }
    }
  }
}
```

```
JTextArea.appendJTextArea.appendtextArea.append(line)synchronizedruntextArea.
append(line)appendText(line)
  public synchronized void appendText(line){
    textArea.append(line);
  }
synchronizedtextAreatextArea
```

The finalize() method is called by the Java virtual machine (JVM)* before the program exits to give the program a chance to clean up and release resources. Multi-threaded programs should close all Files and Sockets they use before exiting so they do not face resource starvation. The call to server.close() in the finalize() method closes the Socket connection used by each thread in this program.

```
  protected void finalize(){
//Objects created in run method are finalized when
//program terminates and thread exits
     try{
        server.close();
    } catch (IOException e) {
        System.out.println("Could not close socket");
        System.exit(-1);
    }
  }
```

Example : Client-Side Program

```
import java.awt.Color;

import java.awt.BorderLayout;

import java.awt.event.*;

import javax.swing.*;



import java.io.*;

import java.net.*;



class SocketClient extends JFrame

                 implements ActionListener {



  JLabel text, clicked;

  JButton button;

  JPanel panel;

  JTextField textField;

  Socket socket = null;

  PrintWriter out = null;
```

```
   BufferedReader in = null;


   SocketClient(){ //Begin Constructor
     text = new JLabel("Text to send over socket:");
     textField = new JTextField(20);
     button = new JButton("Click Me");
     button.addActionListener(this);


     panel = new JPanel();
     panel.setLayout(new BorderLayout());
     panel.setBackground(Color.white);
     getContentPane().add(panel);
     panel.add("North", text);
     panel.add("Center", textField);
     panel.add("South", button);
   } //End Constructor


  public void actionPerformed(ActionEvent event){
     Object source = event.getSource();


     if(source == button){
//Send data over socket
        String text = textField.getText();
```

```
          out.println(text);
              textField.setText(new String(""));
//Receive text from server
       try{
              String line = in.readLine();
          System.out.println("Text received :" + line);
       } catch (IOException e){
             System.out.println("Read failed");
             System.exit(1);
       }
     }
  }

  public void listenSocket(){
//Create socket connection
     try{
       socket = new Socket("VEER-PC", 4444);
       out = new PrintWriter(socket.getOutputStream(), true);
       in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
     } catch (UnknownHostException e) {
       System.out.println("Unknown host: kq6py.eng");
       System.exit(1);
     } catch  (IOException e) {
       System.out.println("No I/O");

       System.exit(1);

     }

  }


   public static void main(String[] args){

        SocketClient frame = new SocketClient();

            frame.setTitle("Client Program");

        WindowListener l = new WindowAdapter() {

                public void windowClosing(WindowEvent e) {

                        System.exit(0);

                }

        };


        frame.addWindowListener(l);

        frame.pack();

        frame.setVisible(true);

            frame.listenSocket();

  }

}
```

Example : Client-Side Behavior

The client program presents a simple user interface and prompts for text input. When you click the Click Me button, the text is sent to the server program. The client program expects an echo from the server and prints the echo it receives on its standard output.

The client program establishes a connection to the server program on a particular host and port number in its listenSocket method, and sends the data entered by the end user to the server program in its actionPerformed method. The actionPerformed method also receives the data back from the server and prints it to the command line.

listenSocket Method

The listenSocket method first creates a Socket object with the **computer name** (ex: VEER-PC) and port number (4321) where the server program is listening for client connection requests. Next, it creates a PrintWriter object to send data over the socket connection to the server program. It also creates a BufferedReader object to read the text sent by the server back to the client.

```
public void listenSocket(){
//Create socket connection
   try{
     socket = new Socket("kq6py", 4321);
     out = new PrintWriter(socket.getOutputStream(),
                 true);
     in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
   } catch (UnknownHostException e) {
     System.out.println("Unknown host: kq6py");
     System.exit(1);
   } catch  (IOException e) {
     System.out.println("No I/O");
     System.exit(1);
   }
}
```

actionPerformed Method

The actionPerformed method is called by the Java platform for action events such as button clicks. This actionPerformed method code gets the text in the Textfield object and passes it to the PrintWriter object, which then sends it over the socket connection to the server program.

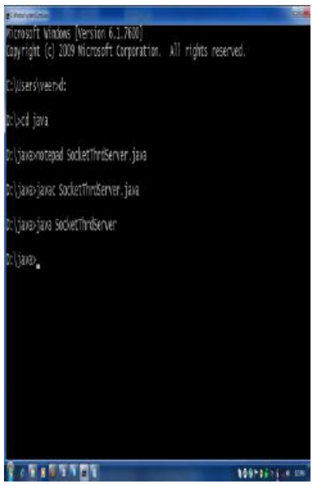
The actionPerformed method then makes the Textfield object blank so it is ready for more end user input. Lastly, it receives the text sent back to it by the server and prints the text out.

```
public void actionPerformed(ActionEvent event){
   Object source = event.getSource();

   if(source == button){
//Send data over socket
      String text = textField.getText();
      out.println(text);
      textField.setText(new String(""));
      out.println(text);
   }
//Receive text from server
   try{
     String line = in.readLine();
     System.out.println("Text received: " + line);
   } catch (IOException e){
     System.out.println("Read failed");
     System.exit(1);
   }
}
```

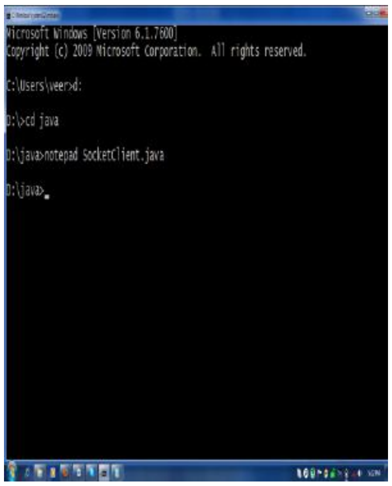
Compile and Run of server and client programs:

To run the example programs, start the server program first. If you do not, the client program cannot establish the socket connection. Here are the compiler and interpreter commands to compile and run the example.

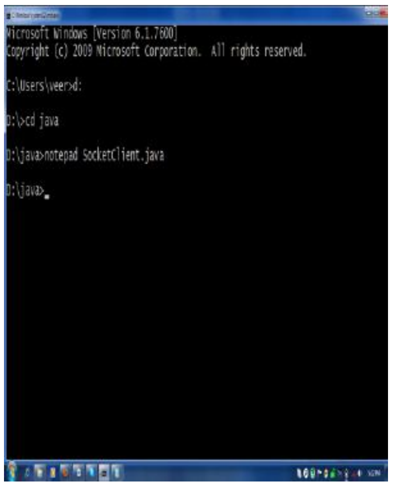
- ❖ Open 1st command Window for Server.
- ❖ Next You can open your desired number of command Windows for clients.



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>notepad SocketThrdServer.java

D:\java>javac SocketThrdServer.java

D:\java>java SocketThrdServer

D:\java>_
```

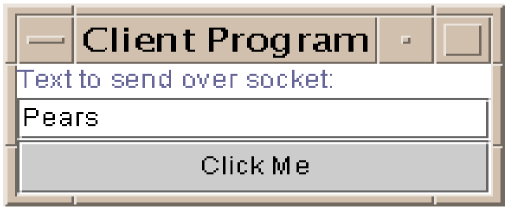


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>notepad SocketClient.java

D:\java>_
```

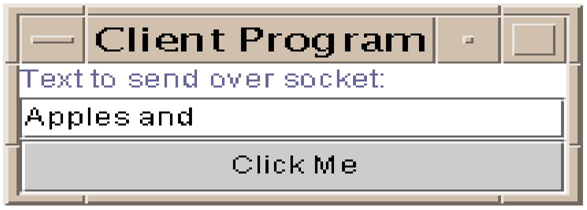


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\veer>d:

D:\>cd java

D:\java>notepad SocketClient.java

D:\java>_
```

You will see the outputs like this:

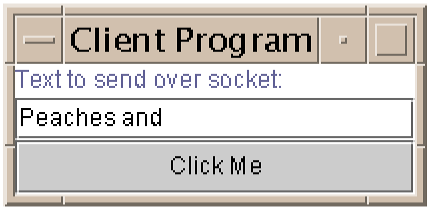


Pears

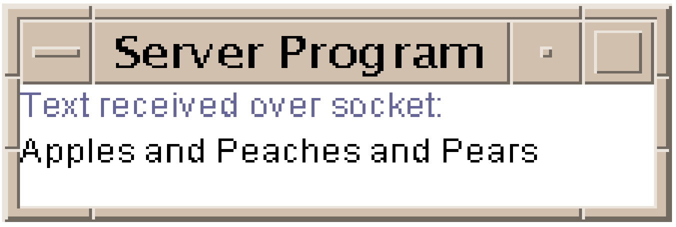


Apples and

Fig:Client3 Fig:Client1



Peaches and



Apples and Peaches and Pears

Fig:Client2

Distributed programming in Java: conventions

Good idea when constructing a server application in a classic client/server architecture (when multiple client connections towards a single server are needed) is to implement one-thread-per-connection technique, where every inbound connection is being delegated to a thread that will take care of every aspect of that connection independently from all other connections. Not only that entire processing is being encapsulated inside that thread, but also normal processing of the main serverâs thread is being enabled because it can (after you delegate the connection to a independent thread) continue waiting for the next inbound connection, and thus not lose any time.

Programmer should think, though, about an efficient number of active thread at one time “ if the server encounters heavy traffic it should stop enabling connections (maybe it as better to deny a service than to give a bad service which may fail at any time).

One more thing programmer should be aware of is the logging process. It is almost impossible to imagine any kind of serious distributed programming without using some logger. Numeroud loggers are available in the market, in Java a wide freeware selection is present (Avalon, JDK>=1.4 has integrated logging), but a definite industry standard for years already is the Apache log4j logger because of its ease of use and wide specter of implemented abilities. Loggers are used primarily because debugging does not help when dealing with distributed applications and also because it allows software specialists to see in details how application worked for example a day or two before when some error occurred. Often program use logging even after development “ when they are put on production servers.

Finally, we should also say a couple of words concerning thread pools. Thread pools are extremely helpful because they help you to work with a number of threads without any risk of having a host thread • (it as a thread that is out of control), and also they help you to control all threads active in the program at every moment " new threads can be added to the pool and removed when they are not useful any more, schedule tasks can be added also (Java TimerTask is not a good solution) and a lot of other things. Since Java 5.0 a very powerful mechanism called Executor framework is integral part of standard Java " it is a very flexible collection of classes and powerful enough for most of real-life situations.

Object serialization

One more thing deeply integrated into Java is serialization. Every object which implements interface java.io.Serializable is said to be exactly that " serializable. It means that that object can be represented like a stream of bytes which can completely describe it. This interface serves just as a sign to the Java compiler " it does not have any methods in it.

Why is the serialization a good thing? Because it allows complete object representation (not only description, but also values of data inside the object) as a unique series of bytes and it allows, at the same time, to reinterpret original object using that serialized byte data.

Basically, because a specific kind of streams can be used to hold serialized objects data, you may save a serialized object into a file, and also (which is much more interesting for us, network programmers) send that stream of bytes to another computer, using TCP for example, so a program that accepts data can make a reverse action on data stream and create an object which will afterwards be in the same state as it was on the source side of the connection. But, keep in mind that the other side (acceptors side) must have a definition of the object so it can have information how to reinterpret the object byte stream (basically, if you wish to send instantiated MyClass object you need the same MyObject.class file on both side of the connection). One more thing which that should be kept in mind all the time is that serialization is a Java specific thing (non-portable) which is very dependable on JDK version (the same Java object may have completely different object stream signature in JDK 1.4 and JDK 6.0).

Object is serialized from memory into a byte stream using ObjectOutputStream class, and object of type ObjectInputStream is used to reinterpret object from a byte data stream. When these object streams are lined to the socketâ€™s underlying data streams you will be able to â€œsend objectsâ€ through the network â€“ which is, you must agree, a very powerful programming technique (RMI uses serialization as a transport mechanism).

Simple example of how to send an object to a server:

```
MyObject msgObject = new MyObject (...);
try {
  socket = new Socket(url, port);
  ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
  oos.writeObject(msgObject);
  oos.flush();
}
catch(IOException e) {
   ...
}
finally {
   ...
}
```

And a simple on-thread server which can receive and reinterpret sent object is:

```
ServerSocket serverSocket = new ServerSocket(port);
Socket socket = null;
MyObject myObject = null;
try {
   socket = serverSocket.accept();
   ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());
   Object obj = reader.readObject();

   if (!(obj instanceof MyObject))
       throw new RuntimeException("Received object is not supported");
   else
       myObject = (MyObject) obj;

   doSomethingWithTheObject(myObject);

}
catch(Exception e) {
   ...
}
finally {
   ...
}
```

Conclusion

In this paper introduction to full spectrum of techniques in java networking are explained. We have seen techniques for programming clients and servers using both TCP and UDP services. If you have previous experience of network programming, you will probably agree that java makes network programming much easier than do most other languages. Thus, one of Java's great strengths is painless networking. As much as possible, the underlying details of networking have been abstracted away.

EXTRA INFO:

Multicasting Through Java

Introduction

Multicasting, is the Internet's version of broadcasting. A site that multicasts information is similar in many ways to a television or radio station that broadcasts its signal. The signal originates from one source, but it can reach everyone in the station's signal area. The information passes on by those who don't want to catch the signal or don't have the right equipment.

Broadcasting is generally suited to any applications that requires a number of machine in a distributed group to receive the same data; for example conferencing, group mail and news distribution, and network management.

Most high-level network protocols only provide a unicast transmission service. That is, nodes of the network only have the ability to send to one other node at a time. All transmission with a unicast service is inherently point-to-point. If a node wants to send the same information to many destinations using a unicast transport service, it must perform a replicated unicast, and send N copies of the data to each destination in turn.

A better way to transmit data from one source to many destinations is to provide a multicast transport service. With a multicast transport service, a single node can send data to many destinations by making just a single call on the transport service:

For those applications which involve a single node sending to many recipients, a multicast facility is clearly a more natural programming paradigm than unicast. When a multicast service is implemented over such a network, there is a huge improvement in performance. If the hardware supports multicast, A packet which is destined for N recipients can be sent as just a single packet!

TOP

Multicast Groups

The notion of *"group"* is essential to the concept of multicasting. By definition a multicast message is sent from a source to a group of destination hosts. In IP multicasting, multicast groups have an ID called multicast group ID. Whenever a multicast message is sent out, a multicast group ID specifies the destination group. These group ID's are essentially a set of IP addresses called "Class D" explained in the following section. Therefore, if a host (a process in a host) wants to receive a multicast message sent to a particular group, it needs to somehow listens to all messages sent to that particular group. Different Ipv4 addresses are explained in the following section.

TOP

Different types of IPv4 Addresses

There are three types of IPv4 addresses: unicast, broadcast, and multicast.

Unicast addresses are used for transmitting a message to a single destination node.

Broadcast addresses are used when a message is supposed to be transmitted to all nodes in a subnetwork.

For delivering a message to a group of destination nodes which are not necessarily in the same subnetwork,*multicast* addresses are used.

Class A, B, and C IP addresses are used for unicast messages, whereas as class D IP address, those in the range 224.0.0.1 to 239.255.255.255, inclusive, and by a standard UDP port number are used for multicast messages.

IP Addresses of Class D - Multicasting

IP addresses of Class D have the following format:

```
Bit no.      0 1 2 3 4 5 6 7 8      16       24       31

Class D 1 1 1 0 |---------multicast address(28)--------|
```

Class D addresses are identified by a one in bit 0,1 and 2 and a zero in bit 3 of the address. This means that 6.25% of all available IP addresses are of this class.

The range of Class D addresses are in dotted decimal notation from 224.h.h.h.h to 239.h.h.h, where h is a number from 0 to 255. Address 224.0.0.0 is reserved and can not be used, while address 224.0.0.1 is used to address all hosts that take part in IP multicasting.

Class D addresses are used for multicasting and does not have a network part and hosts part. IP multicasting makes it possible to send IP datagrams to a group of hosts, which may be spread across many networks.

Life of Multicast Packets (TTL's)

Broadcast packets need to have a finite life inorder to avoid bouncing of the packets around the network forever. Each packet has a time to live (TTL) value, a counter that is decremented every time the packet passes through an hop i.e a router between the network. Because of TTLs, each multicast packet is a ticking time bomb.

Take for example, a TV station where TTLs would be the station's signal area -- the limitation of how far the information can travel. As the packet moved around the company's internal network, its TTL would be notched down every time it passed through an router. When the packet's TTL reached 0, the packet would die and not be passed further. Generally multicast with long TTLs -- perhaps 200 - to guarantee that the information will reach around the world

Java's Support

Java makes the programmer easier by implementing classes in java.net package that facilitates our need i.e multicasting. java.net includes a class called *MulticastSocket.* This kind of socket is used on the client-side to listen for packets that the server broadcasts to multiple clients. The multicast datagram socket class is useful for sending and receiving IP multicast packets. Thus, java.net.*MulticastSocket.* and java.net.*DatagramPacket* together used to implement multicasting in java and makes programming easier.

A MulticastSocket is a (UDP) DatagramSocket, with additional capabilities for joining "groups" of other multicast hosts on the internet. One would join a multicast group by first creating a MulticastSocket with the desired port, then invoking the *joinGroup*(InetAddress groupAddr) method. One can leave a group by using the method *leaveGroup*(InetAddress groupAddr).

TOP

A Simple Server and Client Examples

MulticastSocket class

```
Constructors:
public MulticastSocket() throws IOException
This allows us to Create a multicast socket.

public MulticastSocket(int port) throws IOException
This allows us to Create a multicast socket and
bind it to a specific port.
```

*Methods:*Methods other than the joinGroup(addr) and leaveGroup(addr), this class also provides functions, that set and gets the time-to-live for multicast packets sent out on the socket. The TTL sets the IP time-to-live for DatagramPackets sent to a MulticastGroup, which specifies how many "hops" that the packet will be forwarded on the network before it expires. They are setTTl(ttl) and getTTl(ttl). The parameter 'ttl' is an unsigned 8-bit quantity.

All the methods of this class throws IOEXception.

Limitations:Currently applets are not allowed to use multicast sockets.

A Simple Example that explains multicasting

The Following source code is an simple example for broadcasting. It broadcast date and time to its multiple clients.

Server Program

```
import java.net.*;
import java.io.*;
import java.util.*;
public class BroadcastServer
{
```

```
public static final int PORT = 1200;
public static void main(String args[])
throws Exception{ MulticastSocket socket;
DatagramPacket packet; InetAddress address;
address = InetAddress.getByName();
socket = new MulticastSocket();
// join a Multicast group and send the
group salutations socket.joinGroup(address);
byte[] data = null;
for(;;)
{
 Thread.sleep(1000);
 System.out.println("Sending ");
 String str = (new Date()).toString();
 data = str.getBytes();
 packet = new DatagramPacket
 (data,str.length(),address,PORT);
 // Sends the packet socket.send(packet);
 }
 //
 for
 }
 // main
 } // class BroadcastServer
```

Client program

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

 public class BroadcastClient{
 public static final int PORT = 1200;
 public static void main(String args[])
 sthrows Exception{

 MulticastSocket socket;
 DatagramPacket packet;
 InetAddress address;

 address = InetAddress.getByName(args[0]);
 socket = new MulticastSocket(BroadcastServer.PORT);

 //join a Multicast group and send the
 group salutations

 socket.joinGroup(address);
 byte[] data = null;
 packet = new DatagramPacket(data,data.length);

 for(;;)
 {
   // receive the packets
   socket.receive(packet);
   String str = new String(packet.getData());
   System.out.println(" Time signal received from"+
   packet.getAddress() + " Time is : " +str);
 }  // for
 }  // main
}  // class Broadcast
```

Running the Server and Client

First you will need to ensure that a name is associated with a suitable multi-cast address. This requires access to - and understanding of your system's configuration. Add a suitable name, such as BroadTest and assign it the address between 224.0.0.1 to 239.255.255.255.

Now run the server program by issuing the command:

```
Java BroadcastServer BroadTest.
```

You will see a message "Sending " appear at one-second intervals. This suggests that a server is running.

Run the client side by issuing the command:

```
Java BrodcastClient BroadTest.
```

TOP

You should now see the date and time printed at regular intervals on the client machine

Advantages of Multicasting

In many cases the multicasting capability is desirable because of some following advantages.

The first major advantage of using multicasting is the *decrease of the network load.* There are many applications like stock ticker applications which are required to transmit packets to hundreds of stations. The packets sent to these stations share a group of links on their paths to their destinations. Since multicasting requires the transmission of only a single packet by the source and replicates this packet only if it is necessary multicast transmission can conserve the so much needed network bandwidth.

Another place where multicasting can be very helpful is in *resource discovery.* There are many applications in which a host needs to find out whether a certain type of service is available or not. Using multicast messages and sending the query to those hosts which are potentially capable of providing this service would be of great help. Although some applications use multicast messages to transmit a packet to a group of hosts residing on the same network, there is no reason to impose this limitation. The scope of multicast packets can be limited by using the time-to-live (TTL) field of these packets.

Another important feature of multicasting is its support for *datacasting* applications. In recent years, multimedia transmission has become more and more popular. The audio and video signals are captured, compresses and transmitted to a group of receiving stations. Instead of using a set of point-to-point connections between the participating nodes, multicasting can be used for distribution of the multimedia data to the receivers. In real world stations may join or leave an audio-cast or a video-cast at any time.

Another feature of multicasting is *flexibility* in joining and leaving a group provided by multicasting can make the variable membership much easier to handle.

MBONE (Multicast Bone)

A brief explanation of the MBONE network is given in this section.

Today, the MBONE is a critical piece of the technology that's needed to make multiple person data voice, and video conferencing on the Internet -- in fact, sharing any digital information -- cheap and convenient. Unfortunately, the majority of the routers on the Internet today don't know how to handle multicasting. Most routers are set up to move traditional Internet Protocol (IP) **unicast packets** -- information that has a single, specific destination. Most routers today are **unicast routers:** They are designed to move information from a specific place to another specific place. However, routers that include multicasting capabilities are becoming more common.

So in order to facilitate multicasting MBONE network was created. It's a *"**virtual network**"* - a network that runs on top of the Internet and it allows multicast packets to traverse the Net. The MBONE is called a virtual network because it shares the same physical media -- wires, routers and other equipment -- as the Internet.

The MBONE allows multicast packets to travel through routers that are set up to handle only unicast traffic. Software that utilizes the MBONE hides the multicast packets in traditional unicast packets so that unicast routers can handle the information.The scheme of moving multicast packets by putting them in regular unicast packets is called tunneling. In the future, most commercial routers will support multicasting, eliminating the headaches of tunneling information through unicast routers.

Machines (workstations or routers) that are equipped to support multicast IP are called mrouters (multicast routers). Mrouters are either commercial routers that can handle multicasting or (more commonly) dedicated workstations running special software that works in conjunction with standard routers. *Multicasting is a network routing facility -- a method of sending packets to more than one site at a time.* The MBONE is a loose confederation of sites that currently implement IP multicasting.

MBONE Advantages

You can use the MBONE to do the following:

- Collaborate with colleagues in real time by using a shared virtual "whiteboard"
- Hear and see live lectures from famous professors or scientists, and even ask them questions
- Listen to radio stations that "broadcast" on the Internet

- Start your own radio show
- See live pictures of spacebound NASA astronauts on the space shuttle
- Attend a virtual poetry reading where you hear the words in the author's own voice
- See and hear rehearsals of Silicon Valley garage bands
- Attend an Internet Engineering Task Force meeting without leaving your office

In the future, the MBONE may make it possible for you to do the following:

- Engage 5,000 other people in a huge intercontinental computer game
- See reruns of "Gilligan's Island" and share your snide comments in real time with faraway friends
- Put your own garage band's rehearsals online for all to see (and hear)
- Automatically download and install authorized upgrades and bug-fixes to your computer software, without your intervention
- "Chat" in real time with 20 other users (as you can with Internet Relay Chat, except that you'll use your voice instead of your overworked fingers)
- Do plenty of other things that haven't been thought of yet

Conclusion

In this paper, we first reviewed why and for what applications multicasting is important. Then, the essential concept of group membership was discussed. Here a sample program in java for broadcasting shows the easiness to implement such a complex muticasting. Thus java with it's MulticastSocket class and methods provides the easier way to implement the multicasting by abstracting the underlying details of networking. It also introduces the concept of MBONE and its advantages.