**Chapter 4: Networking Basics**

**Networking Basics**

Ken Thompson and Dennis Ritchie developed UNIX in concert with the C language atBell Telephone Laboratories, Murray Hill, New Jersey, in 1969. For many years, thedevelopment of UNIX remained in Bell Labs and in a few universities and researchfacilities that had the DEC PDP machines it was designed to be run on. In 1978, Bill Joywas leading a project at Cal Berkeley to add many new features to UNIX, such as virtualmemory and full-screen display capabilities. By early 1984, just as Bill was leaving tofound Sun Microsystems, he shipped 4.2BSD, commonly known as *Berkeley UNIX.*

4.2BSD came with a fast file system, reliable signals, interprocess communication, and,most important, networking. The networking support first found in 4.2 eventually becamethe de facto standard for the Internet. Berkeley's implementation of TCP/IP remains theprimary standard for communications within the Internet. The socket paradigm forinterprocess and network communication has also been widely adopted outside ofBerkeley. Even Windows and the Macintosh started talking "Berkeley sockets" in the late'80s.

**Socket Overview:**

A network **socket** is same like an electrical socket.Various plugs around the network have a standard way of deleivering their payload.Anything that understands the standard protocol can plug-in to the socket & communicate.

A **port** is a software address on a computer on the network.For instance the news server is a piece of software that is normally addressed through port 119,the pop server through port 110 & so on.

A **socket** is a communication path to a port when you want your program to communicate over the network, you have to give it a way of addressing the port .

Basically,

**Socket=IP+Port no.**

**Client/Server:**

A ***server***is anything that has some resource that can be shared. There are

*compute servers,* which provide computing power; *print servers,* which manage a

collection of printers; *disk servers,* which provide networked disk space; and *web servers,*which store web pages.

A ***client***is simply any other entity that wants to gain access to aparticular server. The interaction between client and server is just like the interactionbetween a lamp and an electrical socket. The power grid of the house is the server, andthe lamp is a power client. The server is a permanently available resource, while theclient is free to "unplug" after it is has been served.

In Berkeley sockets, the notion of a socket allows a single computer to serve many

different clients at once, as well as serving many different types of information. This featis managed by the introduction of a *port,* which is a numbered socket on a particularmachine. A server process is said to "listen" to a port until a client connects to it. A serveris allowed to accept multiple clients connected to the same port number, although eachsession is unique. To manage multiple client connections, a server process must bemultithreaded or have some othermeans of multiplexing the simultaneous I/O.

**Reserved Sockets**

Once connected, a higher-level protocol ensues, which is dependent on which port youare using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these willseem familiar to you if you have spent any time surfing the Internet. Port number 21 is forFTP, 23 is for Telnet, 25 is for e-mail, 79 is for finger, 80 is for HTTP, 119 is fornetnews—and the list goes on. It is up to each protocol to determine how a client shouldinteract with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer

hypertext pages and images. It is quite a simple protocol for a basic page-browsing webserver. Here's how it works. When a client requests a file from an HTTP server, an actionknown as a *hit,* it simply prints the name of the file in a special format to a predefined portand reads back the contents of the file. The server also responds with a status codenumber to tell the client whether the request can be fulfilled and why.

Here's an exampleof a client requesting a single file, **/index.html**, and the server replying that it hassuccessfully found the file and is sending it to the client:

**Server C lient**

Listens to port 80. Connects to port 80.

Accepts the connection. Writes "GET /index.html HTTP/1.0\\n\\n".

Reads up until the second end-of line

(\\n).

Sees that GET is a known

command and that HTTP/1.0 is a

valid protocol version.

Reads a local file called /index.html.

Writes "HTTP/1.0 200 OK\\n\\n". " 200" means "here comes the file."

Copies the contents of the file into

the socket. Reads the contents of the file and displays it.

Hangs up. Hangs up.

**Proxy Servers**

A ***proxy server***speaks the client side of a protocol to another server. This is often

required when clients have certain restrictions on which servers they can connect to.

Thus, a client would connect to a proxy server, which did not have such restrictions, and

the proxy server would in turn communicate for the client. A proxy server has the

additional ability to filter certain requests or cache the results of those requests for future

use. A caching proxy HTTP server can help reduce the bandwidth demands on a local

network's connection to the Internet. When a popular web site is being hit by hundreds of

users, a proxy server can get the contents of the web server's popular pages once,

saving expensive internetwork transfers while providing faster access to those pages to

the clients.

**Internet Addressing**

Every computer on the Internet has an *address.* An **Internet address** is a number that

uniquely identifies each computer on the Net. There are 32 bits in an IP address, and we

often refer to them as a sequence of four numbers between 0 and 255 separated by dots

(.). This makes them easier to remember, because they are not randomly assigned—they

are hierarchically assigned. The first few bits define which class of network, lettered A, B,

C, D, or E, the address represents. Most Internet users are on a class C network, since

there are over two million networks in class C. The first byte of a class C network is

between 192 and 224, with the last byte actually identifying an individual computer

among the 256 allowed on a single class C network. This scheme allows for half a billion

devices to live on class C networks.

**Domain Naming Service (DNS)**

The Internet wouldn't be a very friendly place to navigate if everyone had to refer to their

addresses as numbers. For example, it is difficult to imagine seeing "http://192.9.9.1/" at

the bottom of an advertisement. Thankfully, a clearinghouse exists for a parallel hierarchy

of names to go with all these numbers. It is called the ***Domain Naming Service (DNS).***Just

as the four numbers of an IP address describe a network hierarchy from left to right, the

name of an Internet address, called its *domain name,* describes a machine's location in a

name space, from right to left. For example, **www.starwave.com** is in the COM domain

(reserved for U.S. commercial sites), it is called starwave (after the company name), and

www is the name of the specific computer that is Starwave's web server. www corresponds

to the rightmost number in the equivalent IP address.

**The Networking Classes and Interfaces**

The classes contained in the **java.net** package are listed here:

Authenticator (Java 2) JarURLConnection (Java 2) SocketPermission

ContentHandler MulticastSocket URL

DatagramPacket NetPermission URLClassLoader (Java 2)

DatagramSocket PasswordAuthentication(Java 2)URLConnection

DatagramSocketImpl ServerSocket URLDecoder (Java 2)

HttpURLConnection Socket URLEncoder

InetAddress SocketImpl URLStreamHandler

The **java.net**package's interfacesare listed here:

ContentHandlerFactory SocketImplFactory URLStreamHandlerFactory

FileNameMap SocketOptions

**InetAddress**

The **InetAddress** class is used to encapsulateboth the numerical IP address and the domain name for thataddress. You interact with this class by using the name of an IP host, which is moreconvenient and understandable than its IP address. The **InetAddress** class hides the

number inside.

**Factory Methods**

The **InetAddress** class has no visible constructors. To create an **InetAddress** object,

you have to use one of the available factory methods. *Factory methods* are merely a

convention whereby static methods in a class return an instance of that class.

In the case of **InetAddress**, the threemethods **getLocalHost( )**, **getByName( )**, and **getAllByName( )** can be used to createinstances of **InetAddress**.

These methods are shown here:

static InetAddress getLocalHost( )throws UnknownHostException

static InetAddress getByName(String *hostName*)throws UnknownHostException

static InetAddress[ ] getAllByName(String *hostName*)throws UnknownHostException

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the

local host. The **getByName( )** method returns an **InetAddress** for a host name passed to

it. If these methods are unable to resolve the host name, they throw an

**UnknownHostException**.

On the Internet, it is common for a single name to be used to represent severalmachines. In the world of web servers, this is one way to provide some degree of scaling.

The **getAllByName( )** factory method returns an array of **InetAddress**es that represent

all of the addresses that a particular name resolves to. It will also throw an

**UnknownHostException** if it can't resolve the name to at least one address.

**Example:**

import java.net.\*;

class inetadd

{

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

InetAddress i = InetAddress.getLocalHost();

System.out.println(i);

i= InetAddress.getByName("shree-PC");

System.out.println(i);

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

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

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

}

}

**Instance Methods**

The **InetAddress** class also has a few nonstatic methods, listed here, which can be used

on the objects returned by the methods just discussed:

boolean equals(Object *other*) Returns **true** if this object has the same Internet

address as *other*.

byte[ ] getAddress( ) Returns a four-element byte array that represents

the object's Internet 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.

int hashCode( )Returns the hash code of the invoking object.

boolean isMulticastAddress( )Returns **true** if this Internet 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.

Internet addresses are looked up in a series of hierarchically cached servers. That means

that your local computer might know a particular name-to-IP-address mapping

automatically, such as for itself and nearby servers. For other names, it may ask a local

DNS server for IP address information. If that server doesn't have a particular address, it

can go to a remote site and ask for it. This can continue all the way up to the root server,

called InterNIC (internic.net). This process might take a long time, so it is wise to structure

your code so that you cache IP address information locally rather than look it up

repeatedly.

**TCP/IP Client Sockets**

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to- point,

stream-based connections between hosts on the Internet. A socket can be used to

connect Java's I/O system to other programs that may reside either on the local machine

or on any other machine on the Internet.

**Note:** Applets may only establish socket connections back to the host from which

the applet was downloaded. This restriction exists because it would be

dangerous for applets loaded through a firewall to have access to any

arbitrary machine.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for

clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to

connect before doing anything. The **Socket** class is designed to connect to server

sockets and initiate protocol exchanges.

The creation of a **Socket** object implicitly establishes a connection between the client and

server. There are no methods or constructors that explicitly expose the details of

establishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName*, int*port*)Creates a socket connecting the local host to the

named host and port; can throw an**UnknownHostException** or an **IOException**.

Socket(InetAddress *ipAddress*,int *port*)Creates a socket using a preexisting **InetAddress**

object and a port; can throw an **IOException**.

A socket can be examined at any time for the address and port information associated

with it, by use of the following methods:

InetAddress getInetAddress( )Returns the **InetAddress** associated with the**Socket** object.

int getPort( )Returns the remote port to which this **Socket** objectis connected.

int getLocalPort( )Returns the local port to which this **Socket** object isconnected.

Once the **Socket** object has been created, it can also be examined to gain access to the

input and output streams associated with it. Each of these methods can throw an

**IOException** if the sockets have been invalidated by a loss of connection on the Net.

These streams are used to send andreceive data.

InputStream getInputStream( ) Returns the **InputStream** associated with the

invoking socket.

OutputStream getOutputStream()Returns the **OutputStream** associated with the

invoking socket.

void close( )Closes both the **InputStream** and **OutputStream**.

**TCP/IP Server Sockets**

The **ServerSocket** class is used to create servers that listen foreither local or remote client programs to connect to them on published ports. Since theWeb is driving most of the activity on the Internet, this section develops an operationalweb (http) server.

**ServerSocket**s are quite different from normal **Socket**s. When you create a

**ServerSocket**, it will register itself with the system as having an interest in client

connections. The constructors for **ServerSocket** reflect the port number that you wish to

accept connections on and, optionally, how long you want the queue for said port to be.

The queue length tells the system how many client connections it can leave pending

before it should simply refuse connections. The default is 50. The constructors might

throw an **IOException** under adverse conditions. Here are the constructors:

ServerSocket(int *port*)Creates server socket on the specified port with a

queue length of 50.

ServerSocket(int *port*, int*maxQueue*)Creates a server socket on the specified port with a

maximum queue length of *maxQueue.*

ServerSocket(int *port*, int*maxQueue*,InetAddress*localAddress*)

Creates a server socket on the specified port with amaximum queue length of *maxQueue.* On amultihomed host, *localAddress* specifies the IPaddress to which this socket binds.

**ServerSocket** has one additional method called **accept( )**, which is a blocking call that will

wait for a client to initiate communications, and then return with a normal **Socket** that is

then used for communication with the client.

**Example:**

1. //program for tcp/ip client server client send no to server displays its factorial.

import java.net.\*;

import java.io.\*;

class clientfact

{

public static void main(String ar[])throws Exception

{

Socket s=new Socket("atharv.",3000);

System.out.println("Connected to:"+s.getInetAddress());

BufferedReader br=new BufferedReader(new InputStreamReader(s.getInputStream()));

PrintWriter w=new PrintWriter(s.getOutputStream(),true);

BufferedReader br1=new BufferedReader(new InputStreamReader(System.in));

String msg=br1.readLine();

w.println(msg);

msg=br.readLine();

System.out.println("no' is"+msg);

s.close();

}

}

import java.net.\*;

import java.io.\*;

class serverfact

{

public static void main(String ar[])throws Exception

{

ServerSocket p=new ServerSocket(3000);

Socket s=p.accept();

System.out.println("Listened to:"+s.getInetAddress());

BufferedReader br=new BufferedReader(new InputStreamReader(s.getInputStream()));

PrintWriter w=new PrintWriter(s.getOutputStream(),true);

BufferedReader br1=new BufferedReader(new InputStreamReader(System.in));

String msg=br.readLine();

System.out.println("no from client is:"+msg);

int i=Integer.parseInt(msg);

int f=1;

for(int j=1;j<=i;j++)

{

f=f\*j;

}

msg="factorial is:"+f;

w.println(msg);

s.close();

p.close();

}

}

1. //program for contineous chatting between client and server untill any one says end

import java.io.\*;

import java.net.\*;

import java.awt.\*;

import java.awt.event.\*;

class clientchat extends Frame implements ActionListener

{

TextField f=new TextField(30);

static TextArea a=new TextArea();

static Socket s;

static String msg="";

static BufferedReader br;

PrintWriter w;

public clientchat()

{

add(f,BorderLayout.NORTH);

add(a,BorderLayout.CENTER);

f.addActionListener(this);

try

{

s=new Socket("127.0.0.1",6000);

br=new BufferedReader(new InputStreamReader(s.getInputStream()));

w=new PrintWriter(s.getOutputStream(),true);

}

catch(Exception e){}

}

public void actionPerformed(ActionEvent e)

{

if(e.getSource()==f)

{msg=f.getText();

w.println(msg);

if(msg.equals("end"))

{

System.exit(0);

}

f.setText("");

}}

public static void main(String ar[])throws Exception

{

clientchat t=new clientchat();

t.setSize(400,400);

t.setVisible(true);

t.setTitle("Client");

t.addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}});

while(true)

{

msg=br.readLine();

a.append("\nServer:"+msg);

if(msg.equals("end"))

{

break;

}

}

}

}

import java.io.\*;

import java.net.\*;

import java.awt.\*;

import java.awt.event.\*;

class serverchat extends Frame implements ActionListener

{

TextField f=new TextField(30);

static TextArea a=new TextArea();

static Socket s;

static String msg="";

static BufferedReader br;

PrintWriter w;

public serverchat()

{ setSize(400,400);

setVisible(true);setTitle("Server");

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}});

add(f,BorderLayout.NORTH);

add(a,BorderLayout.CENTER);

f.addActionListener(this);

try

{

ServerSocket p=new ServerSocket(6000);

s=p.accept();

br=new BufferedReader(new InputStreamReader(s.getInputStream()));

w=new PrintWriter(s.getOutputStream(),true);

}

catch(Exception e){}

}

public void actionPerformed(ActionEvent e)

{

if(e.getSource()==f)

{msg=f.getText();

w.println(msg);

if(msg.equals("end"))

{

System.exit(0);

}

}

f.setText("");

}

public static void main(String ar[])throws Exception

{

serverchat t=new serverchat();

while(true)

{

msg=br.readLine();

a.append("\nClient:"+msg);

if(msg.equals("end"))

{

break;

}

}

}

}

[**URL:-**](URL:-)

URL stands for Uniform resource locator.

The URL provides a reasonably intelligible form to uniquely identify or address information on the internet.

All URLs share the same basic format .

URL has four parts.

**Protocol name Host name Port no File name**

**1 2 3 4**

Ex:

<http://www.osborne.com:80/index.html>

The URL class has following constructors

URL(String urlspecifier)throws MalformedURLException

URL(String protocolname,String hostname,int port,String path) throws MalformedURLException

URL(String protocolname,String hostname,String path) throws MalformedURLException

Methods of URL

1. String getProtocol()🡪 returns protocol name
2. String getHost()🡪 returns host name
3. String getFile()🡪 returns file name
4. int getPort()🡪returns port number
5. String toExternalForm()🡪 returns string equivalent of URL
6. URLConnection openConnection()🡪 is used to access the actual content information of a URL.

**Example:**

import java.net.\*;

class URLDemo

{

public static void main(String ar[])throws MalformedURLException

{

URL u=new URL(“<http://www.google.com:80/index.html>”);

System.out.println(“Protocol is”+u.getProtocol());

System.out.println(“Host is”+u.getHost());

System.out.println(“Port is”+u.getPort());

System.out.println(“File is”+u.getFile());

}

}

**URI class:**

This class provides methods for creating URI instances from its components or by parsing the string form of those components, for accessing and retrieving different components of a URI instance.  
**What is URI?**  
URI stands for Uniform Resource Identifier. A Uniform Resource Identifier is a sequence of characters used for identification of a particular resource. It enables for the interaction of the representation of the resource over the network using specific protocols.

**URI and URL**

The difference between them is straightforward after knowing their definitions:

* **Uniform Resource Identifier (URI) −** a sequence of characters that allows the complete identification of any abstract or physical resource
* **Uniform Resource Locator (URL) −** a subset of URI that, in addition to identifying where a resource is available, describes the primary mechanism to access it

### Syntax

Every URI, regardless if it's a URL or not, follows a particular form:

|  |  |
| --- | --- |
| 1 | scheme:[//authority][/path][?query][#fragment] |

Where each part is described as follows:

* **scheme** − for URLs, is the name of the protocol used to access the resource, for other URIs, is a name that refers to a specification for assigning identifiers within that scheme
* *authority* − an optional part comprised of user authentication information, a host and an optional port
* **path** − it serves to identify a resource within the scope of its *scheme* and *authority*
* **query** − additional data that, along with the *path,* serves to identify a resource. For URLs, this is the query string
* **fragment** − an optional identifier to a specific part of the resource

**To easily identify if a particular URI is also a URL, we can check its scheme**. Every URL has to start with any of these schemes: *ftp*, *http*, *https,* *gopher*, *mailto*, *news*, *nntp*, *telnet*, *wais*, *file*, or *prospero*. If it doesn't start with it, then it's not a URL.

**Example of URI**

abc://admin:admin@geeksforgeeks.org:1234/path/data?key=value&key2=value2#fragid1

Scheme Authority Path Query Fragment

**URLConnection:-**

URLConnection is a general-purpose class for accessing the attributes of a remote

resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally.

These attributes are exposed by the HTTP protocol specification and, as such, only

make sense for URL objects that are using the HTTP protocol.

The methods used for getting remote resource information are as follows,

1)int getContentLength()🡪returns the size in bytes of the content associated with the resource.if length is unavailable -1 is returned.

2)String getContentType()🡪 returns type of content found in the resource,if type is unavailable it returns null.

Ex: text/html

3)long getDate()🡪 returns the time and date of the response in terms of milliseconds.

4)long getExpiration()🡪 returns the expiration time and date of the resource in terms of milliseconds since jan 1,1970GMT.

5)long getLastModified()🡪returns the time and date of last modification of the resource in terms of milliseconds since jan 1,1970GMT.Zero is returned if the date is unavailable.

6)InputStream getInputStream() throws IOException🡪returns an InputStream that is linked to the resource. This stream is used to obtain the contents.

**Example:**

import java.net.\*;

import java.io.\*;

import java.awt.event.\*;

import java.awt.\*;

class urlconn extends Frame implements ActionListener

{TextField f=new TextField();

TextArea a=new TextArea();

String s="";

public urlconn()

{

add(f,BorderLayout.NORTH);

add(a,BorderLayout.CENTER);

f.addActionListener(this);

setSize(400,400);

}

public void actionPerformed(ActionEvent e)

{

if(e.getSource()==f)

{

try

{

URL u=new URL(f.getText());

URLConnection con=u.openConnection();

BufferedReader br=new BufferedReader(new InputStreamReader(con.getInputStream()));

int len=con.getContentLength();

if(len!=0)

{

while((s=br.readLine())!=null)

a.append("\n"+s);

br.close();

}

else

a.append("\n"+"content unavailable");

}catch(Exception ae){}

}

}

public static void main(String ar[])throws Exception

{

urlconn c=new urlconn();

c.setVisible(true);

}

}

**Datagram:**

TCP/IP provides reliability of data but it include many complicated algorithms for dealing with congestion control. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. 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.

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.

**DatagramSocket**

DatagramSocket defines four public constructors.

They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int port) throws SocketException

DatagramSocket(int port, InetAddress ipAddress) throws SocketException

DatagramSocket(SocketAddress address) throws SocketException

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

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

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

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

void receive(DatagramPacket packet) throws IOException

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

**Other Methods**

**InetAddress getInetAddress( ):-**If the socket is connected, then the address is returned.Other wise, null is returned.

**int getLocalPort( ):-**Returns the number of the local port.

**int getPort( ):-**Returns the number of the port to which the socket isconnected.It returns –1 if the socket is not connectedto a port.

**boolean isBound( )**:-Returnstrueif the socket is bound to an address.Returnsfalseother wise.

**boolean isConnected( )**:-Returnstrueif the socket is connected to a ser ver.Returnsfalseother wise.

**void setSoTimeout(int millis)throws SocketException**:- Sets the time-out period to the number of millisecondspassed inmillis.

**DatagramPacket**

DatagramPacket defines several constructors.

Four are shown here:

DatagramPacket(byte data[ ], int size)

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

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

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

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.

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

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.

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 tosize.

**void setLength(int size):-**Sets the length of the packet to size.

**void setPort(int port)**:-Sets the port to port

**Examples:**

**ClientProgram**

import java.net.\*;

import java.io.\*;

class gramclient

{

public static void main(String ar[])throws Exception

{

int clport=2000;

int serport=3000;

byte buf[]=new byte[1024];

DatagramSocket s=new DatagramSocket(clport);

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

DatagramPacket p;

InetAddress i=InetAddress.getByName("localhost");

String str=br.readLine();

buf=str.getBytes();

p=new DatagramPacket(buf,0,buf.length,i,serport);

s.send(p);

buf=new byte[1024];

p=new DatagramPacket(buf,0,buf.length);

s.receive(p);

str=new String(p.getData(),0,p.getLength());

System.out.println("server says:"+str);

}

}

**ServerProgram**

import java.net.\*;

import java.io.\*;

class gramserver

{

public static void main(String ar[])throws Exception

{

int clport=2000;

int serport=3000;

String str;

byte buf[]=new byte[1024];

DatagramSocket s=new DatagramSocket(serport);

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

DatagramPacket p;

InetAddress i=InetAddress.getByName("localhost");

p=new DatagramPacket(buf,0,buf.length);

s.receive(p);

str=new String(p.getData(),0,p.getLength());

System.out.println("client says:"+str);

buf=new byte[1024];

str=br.readLine();

buf=str.getBytes();

p=new DatagramPacket(buf,0,buf.length,i,clport);

s.send(p);

}

}