
UNIT 3 NETWORKING FEATURES

Structure	Page Nos.
3.0 Introduction	55
3.1 Objectives	55
3.2 Socket Overview	55
3.3 Reserved Parts and Proxy Servers	59
3.4 Internet Addressing: Domain Naming Services (DNS)	60
3.5 JAVA and the net: URL	61
3.6 TCP/IP Sockets	64
3.7 Datagrams	66
3.8 Summary	69
3.9 Solutions/ Answers	69

3.0 INTRODUCTION

Client/server applications are need of the time. It is challenging and interesting to develop Client/server applications. Java provides easier way of doing it than other programming such as C. Socket programming in Java is seamless. The java.net package provides a powerful and flexible infrastructure for network programming. Sun.* packages have some classes for networking. In this unit you will learn the java.net package, using socket based communications which enable applications to view networking operations as I/O operation. A program can read from a socket or write to a socket as simply as reading a file or writing to a file.

With datagram and stream sockets you will be developing connection less and connection oriented applications respectively. Going through various classes and interfaces in java. net, will be useful in learning, how to develop networking applications easily. In this unit you will also learn use of stream sockets and the TCP protocol, which is the most desirable for the majority of java programmers for developing networking applications.

3.1 OBJECTIVES

After going though this Unit, you will be able to:

- define socket and elements of java networking;
 - describe the stream and datagram sockets, and their usage;
 - Explain how to implement clients and servers programs that communicates with each other;
 - define reserved sockets and proxy servers;
 - implement Java networking applications using TCP/IP Server Sockets, and
 - implement Java networking applications using Datagram Server Sockets.
-

3.2 SOCKET OVERVIEW

Most of the inter process communication uses the *client server model*. The terms client and server refer to as the two processes, which will be communicating with each other. One of the two processes, the *client*, connects to the other process, the *server*, typically to make a request for information. A good analogy of this type of communication can be a person who makes a railway engineering from other person

Socket is a data structure that maintains necessary information used for communication between client & server. Therefore both end of communication has its own sockets.

may be through phone call: person making enquiry is a client and another person providing information is a server.

Notice that the client needs to know about the existence and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are different for the client and the server, but both involve the basic construct of a *socket*.

Java introduces socket based communications, which enable applications to view networking as if it were file I/O— a program which can read from socket or write to a socket with the simplicity as reading from a file or writing to a file. Java provides *stream sockets and datagram sockets*.

Stream Sockets

Stream Sockets are used to provide a connection-oriented service (i.e. TCP-Transmission Control Protocol).

With stream sockets a process establishes a connection to another process. Once the connection is in place, data flows between processes in continuous streams.

Datagram Sockets

This socket are used to provide a connection-less service, which does not guarantee that packets reach the destination and they are in the order at the destination. In this, individual packets of information are transmitted. In fact it is observed that packets can be lost, can be duplicated, and can even be out of sequence.

In this section you will get answers of some of the most common problems to be addressed in sockets programming using Java. Then you will see some example programs to learn how to write client and server applications.

The very first problem you have to address is “How will you open a socket?” Before reaching to the answer to this question you will definitely think that what type of socket is required? Now the answer can be given as follows:

1. If you were programming a client, then you would open a socket like this:

Socket MyClient;

MyClient = new Socket("Machine_Name", PortNumber);

Where “Machine name” is the server machine you are trying to open a connection to, and “PortNumber” is the number of the port on server, on which you are trying to connect it. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users or slandered services like e-mail, HTTP etc. For example, port number 21 is for FTP, 23 is for TLNET, and 80 is for HTTP. It is a good idea to handle exceptions while creating a new Socket.

Socket MyClient;

```
try {  
    MyClient = new Socket("Machine name", PortNumber);  
}  
catch (IOException e)
```

Port is a unique number association with a socket on a machine. In other word's port is a numbered socket on a machine.

```

{
    System.out.println(e);
}

```

2. If you are programming a server, then this is how you open a socket:

```

ServerSocket MyService;
try {
    MyService = new ServerSocket(PortNumber);
}
catch (IOException e)
{
    System.out.println(e);
}

```

While implementing a server you also need to create a socket object from the `ServerSocket` in order to listen for client and accept connections from clients.

```

Socket clientSocket = null;
try {
    serviceSocket = MyService.accept();
}
catch (IOException e)
{
    System.out.println(e);
}

```

You can notice that for a Client side programming you use the `Socket` class and for the Server side programming you use the `ServerSocket` class.

Now you know how to create client socket and server socket. Now you have to create input stream and output stream to receive and send data respectively.

InputStream

On the client side, you can use the `DataInputStream` class to create an input stream to receive response from the server:

```

DataInputStream input;
try {
    input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e)
{
    System.out.println(e);
}

```

The class `DataInputStream` allows you to read lines of text and Java primitive data types in a portable way. It has methods such as `read`, `readChar`, `readInt`, `readDouble`, and `readLine`. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use `DataInputStream` to receive input from the client:

```

DataInputStream input;
try {
    input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e)

```

```
        {  
            System.out.println(e);  
        }  
    }  
    OutputStream
```

On the client side, you can create an output stream to send information to the server socket using the class `PrintStream` or `DataOutputStream` of `java.io`:

```
PrintStream output;  
try {  
    output = new PrintStream(MyClient.getOutputStream());  
}  
catch (IOException e)  
{  
    System.out.println(e);  
}
```

The class `PrintStream` has methods for displaying textual representation of Java primitive data types. Its `write ()` and `println ()` methods are important here. Also, you can use the `DataOutputStream`:

```
DataOutputStream output;  
try {  
    output = new DataOutputStream(MyClient.getOutputStream());  
}  
catch (IOException e)  
{  
    System.out.println(e);  
}
```

The class `DataOutputStream` allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method `writeBytes ()` is a useful one.

On the server side, you can use the class `PrintStream` to send information to the client.

```
PrintStream output;  
try {  
    output = new PrintStream(serviceSocket.getOutputStream());  
}  
catch (IOException e)  
{  
    System.out.println(e);  
}
```

If you have opened a socket for connection, after performing the desired operations your socket should be closed. You should always close the output and input stream before you close the socket.

On the client side:

```
try  
{  
    output.close();  
    input.close();  
    MyClient.close();  
}  
catch (IOException e)
```

```
{
    System.out.println(e);
}
```

On the server side:

```
try
{
    output.close();
    input.close();
    serviceSocket.close();
    MyService.close();
}
catch (IOException e)
{
    System.out.println(e);
}
```

Check Your Progress 1

- 1) Write a program to show that from the client side you send a string to a server that reverse the string which is displayed on the client side.

.....

.....

.....

- 2) Describe different types of sockets.

.....

.....

.....

- 3) What are Datagram and Stream Protocols?

.....

.....

.....

3.3 RESERVED PORTS AND PROXY SERVERS

Now let us see some reserve ports. As we have discussed earlier, there are some port numbers, which are **reserved** for specific purposes on any computer working as a server and connected to the Internet. Most standard applications and protocols use **reserved** port numbers, such as email, FTP, and HTTP.

When you want two programs to talk to each other across the Internet, you have to find a way to initiate the connection. So at least one of the ‘partners’ in the conversation has to know where to find the other one or in other words the address of other one. This can be done by address (IP number + port number) of the one side to the other.

However, a problem could arise if this address must not be taken over by any other program. In order to avoid this, there are some port numbers, which are reserved for specific purposes on any computer connected to the Internet. Such ports are reserved for programs such as ‘Telnet’, ‘Ftp’ and others. For example, Telnet uses port 23, and

FTP uses port 21. Note that for each kind of service, not only a port number is given, but also a protocol name (usually TCP or UDP).

Two services may use the same port number, provided that they use different protocols. This is possible due to the fact that different protocols have different address spaces: port 23 of a one machine in the TCP protocol address space is not equivalent to port 23 on the same machine, in the UDP protocol address space.

Proxy Servers

A proxy server is a kind of buffer between your computer and the Internet resources you are accessing. They accumulate and save files that are most often requested by thousands of Internet users in a special database, called “cache”. Therefore, proxy servers are able to increase the speed of your connection to the Internet. The cache of a proxy server may already contain information you need by the time of your request, making it possible for the proxy to deliver it immediately. The overall increase in performance may be very high.

Proxy servers can help in cases when some owners of the Internet resources impose some restrictions on users from certain countries or geographical regions. In addition to that, a type of proxy server called anonymous proxy servers can hide your IP address thereby saving you from vulnerabilities concerned with it.

Anonymous Proxy Servers

Anonymous proxy servers hide your IP address and thereby prevent your data from unauthorized access to your computer through the Internet. They do not provide anyone with your IP address and effectively hide any information about you. Besides that, they don’t even let anyone know that you are surfing through a proxy server. Anonymous proxy servers can be used for all kinds of Web-services, such as Web-Mail (MSN Hot Mail, Yahoo mail), web-chat rooms, FTP archives, etc. *ProxySite.com* will provide a huge list of public proxies.

Any web resource you access can gather personal information about you through your unique IP address – your ID in the Internet. They can monitor your reading interests, spy upon you, and according to some policies of the Internet resources, deny accessing any information you might need. You might become a target for many marketers and advertising agencies that, having information about your interests and knowing your IP address as well as your e-mail. They will be able to send you regularly their spam and junk e-mails.

3.4 INTERNET ADDRESSING: DOMAIN NAMING SERVICES (DNS)

You know there are thousands of computers in a network; it is not possible to remember the IP address of each system. Domain name system provides a convenient way of finding computer systems in network based on their name and IP address. Domain name services resolves names to the IP addresses of a machine and vice-versa. Domain name system is a hierarchical system where you have a top-level domain name server sub domain and clients with names & IP address.

When you use a desktop client application, such as e-mail or a Web browser, to connect to a remote computer, your computer needs to resolve the addresses you have entered, into the IP addresses it needs to connect to the remote server. DNS is a way to resolve domain name to IP addresses on a TCP/IP network.

The major components of DNS are:

Domain Name Space,
Domain Name Servers and Resource Records (RR),
Domain Name Resolvers (DNRs).

The Domain Name Space: It is a tree-structured name space that contains the domain names and data associated with the names. For example, *astrospeak.indiatimes.com* is a node within the *indiatimes.com* domain, which is a node in the *com* domain. Data associated with *astrospeak.indiatimes.com* includes its IP address. When you use DNS to find a host address, you are querying the Domain Name Space to extract information.

A Domain Name Server provides information about a subset of the Domain Name Space.

The Domain Name Space for an entity is the name by which the entity is known on the Internet. For example, in the organization shown in, you have two entities with two address spaces. The *Times of India* organization is known on the Internet as the *timesofindia.com* domain, and our sample Internet organization is known as the *indiatimes.com* domain. Hosts at these organizations are known as a host name plus the domain name; for example, *money.timesofindia.com*. Similarly, users in these domains can be found by their e-mail aliases; for example, *abc@indiatimes.com*.

The Domain Name Server points to other Domain Name Servers that have information about other subsets of the Domain Name Space. When you query a Domain Name Server, it returns information if it is an authoritative server for that domain. If the Domain Name Server doesn't have the information, it refers you to a higher level Domain Name Server, which in turn can refer you to another Domain Name Server, until it locates the one with the requested information. In this way, no single server needs to have all the information for every host you might need to contact.

A Domain Name Resolver extracts information from Domain Name Servers so you can use host addresses instead of IP addresses in clients such as a Web browser or a File Transfer Protocol (FTP) client, or with utilities such as ping, tracer, or finger. The DNR is typically built into the TCP/IP implementation on the desktop and needs to know only the IP address of the Domain Name Server. Configuring the DNR on the desktop is usually a matter of filling in the TCP/IP configuration data.

3.5 JAVA AND THE NET: URL

You know each package defines a number of classes, interfaces, exceptions, and errors. The *java.net* package contains these, interfaces, classes, and exceptions: This package is used in programming where you need to know some information regarding the internet or if you want to communicate between two host computers.

Interfaces in *java.net*

ContentHandlerFactory
FileNameMap
SocketImplFactory
URLStreamHandlerFactory

Exceptions in *java.net*

BindException
ConnectException
MalformedURLException
NoRouteToHostException

Classes in *java.net*

ContentHandler
DatagramSocket
DatagramPacket
DatagramSocketImpl
HttpURLConnection
InetAddress

ProtocolException
SocketException
UnknownHostException
UnknownServiceException

MulticastSocket
ServerSocket
Socket
SocketImpl
URL
URLConnection
URLEncoder
URLStreamHandler

URL

URL is the acronym for Uniform Resource Locator. It represent the addresses of resources on the Internet. You need to provide URLs to your favorite Web browser so that it can locate files on the Internet. In other words you can see URL as addresses on letters so that the post office can locate for correspondents URL class is provided in the `java.net` package to represent a URL address.

URL object represents a URL address. The URL object always refers to an absolute URL. You can construct from an absolute URL, a relative URL. URL class provides accessor methods to get all of the information from the URL without doing any string parsing.

You can connect to a URL by calling `openConnection()` on the URL. The `openConnection()` method returns a `URLConnection` object. `URLConnection` object is used for 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 URL Connection

Some URLs, such as many that are connected to cgi-bin scripts, allow you to write information to the URL. For example, if you have to search something, then search script may require detailed query data to be written to the URL before the search can be performed.

Before interacting with the URL you have to first establish a connection with the Web server that is responsible for the document identified by the URL.

Then you can use TCP socket for the connection is constructed by invoking the `openConnection` method on the URL object. This method also performs the name resolution necessary to determine the IP address of the Web server.

The `openConnection` method returns an object of type `URLConnection` to the Web server which is requested by calling `connection` on the `URLConnection` object. For input and output handling for the document identified by the URL `InputStream` and `OutputStream` are used respectively.

Below are the various constructors and methods of URL class of `java.net` package.

```
public URL(String protocol, String host, int port, String file)  
throws MalformedURLException
```

```
public URL(String protocol, String host, String file)  
throws MalformedURLException
```

```
public URL(String spec) throws MalformedURLException  
public URL(URL context, String spec) throws MalformedURLException  
public int getPort()
```



```

public String getFile()
public String getProtocol()
public String getHost()
public String getRef()
public boolean equals(Object obj)
public int hashCode()
public boolean sameFile(URL other)
public String toString()
public URLConnection openConnection() throws IOException
public final InputStream openStream() throws IOException
public static synchronized void setURLStreamHandlerFactory(
URLStreamHandlerFactory factory)

```

In the example program given below, URL of homepage of “rediff.com” is created.

```

import java.net.*;
class URL_Test
{
    public static void main(String[] args) throws MalformedURLException
    {
        URL redURL = new URL("http://in.rediff.com/index.html");
        System.out.println("URI Protocol:"+redURL.getProtocol());
        System.out.println("URI Port:"+redURL.getPort());
        System.out.println("URI Host:"+redURL.getHost());
        System.out.println("URI File"+redURL.getFile());
    }
}

```

Output:
URI Protocol:http
URI Port:-1
URI Host:in.rediff.com
URI File/index.html

Check Your Progress 2

- 1) When should you use Anonymous Proxy Servers?

.....

.....

.....

.....

.....

- 2) Explain various kinds of domain name servers.

.....

.....

.....

.....

- 3) Write a program in Java to know the Protocol, Host, Port, File and Ref of a particular URL using *java.net.URL* package.

.....

.....

.....

TCP/IP is a set of protocols used for communication between different types of computers and networks.

3.6 TCP/IP SOCKETS

TCP/IP refers to two of the protocols in the suite: the *Transmission Control Protocol* and the *Internet Protocol*.

These protocols utilize sockets to exchange data between machines. The TCP protocol requires that the machines communicate with one another in a reliable, ordered stream. Therefore, all data sent from one side must be received in correct order and acknowledged by the other side. This takes care of lost and dropped data by means of acknowledgement and re-transmission. UDP, however, simply sends out the data without requiring knowing that the data be received.

In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

UDP is an unreliable protocol; there is no guarantee that the datagrams you have sent will be put in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be put in the order in which they were sent.

In short, TCP is useful for implementing network services: such as remote login (rlogin, telnet) and file transfer (FTP). These services require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. UDP is often used in implementing client/server applications in distributed systems, which are built over local area networks.

We have already discussed about client and server Sockets in section 3.2 of this Unit. Recall the discussions in section 3.2, two packages `java.net.ServerSocket` and `java.net.Socket` were used to create sockets.

Before we discuss about `Socket` and `ServerSocket` class, it is important to know about `InetAddress` class. Let us see what is `InetAddress` class.

InetAddress

This class is used for encapsulating numerical IP address and domain name for that address.

Because `InetAddress` is not having any constructor, its objects are created by using any of the following three methods

```
static InetAddress getLocalHost() throws UnknownHostException:
    returns the InetAddress object representing local host
static InetAddress getByName() throws UnknownHostException:
    returns the InetAddress object for the host name passed to
    it.
```

```
static InetAddress getAllByName() throws UnknownHostException: returns an array
of InetAddresses representing all the addresses that a particular name is resolves to.
```

Java.net.Socket

Constructors

```
public Socket(InetAddress addr, int port): creates a stream socket and connects it to
the specified port number at the specified IP address
```

`public Socket (String host, int port):` creates a stream socket and connects it to the specified port number at the specified host

Methods:

`InetAddress getAddress() :` Return Inet Address of object associated with Socket

`int getPort() :` Return remote port to which socket is connected

`int getLocalPort() :` Return local port to which socket object is connected.

`public InputStream getInputStream():` Get InputStream associated with Socket

`public OutputStream getOutputStream():` Return OutputStream associated with socket

`public synchronized void close() :` closes the Socket.

Java.net.ServerSocket

Constructors:

`public ServerSocket(int port):` creates a server socket on a specified port with a queue length of 50. A port number 0 creates a socket on any free port.

`public ServerSocket(int port, int QueLen):` creates a server socket on a specified port with a queue length of QueLen.

`public ServerSocket(int port, int QueLen, InetAddress localAdd):` creates a server socket on a specified port with a queue length specified by QueLen. On a multihomed host, localAdd specifies the IP address to which this socket binds.

Methods:

`public Socket accept():` listens for a connection to be made to this socket and accepts it. `public void close():` closes the socket.

Java TCP Socket Example

A Server (web server) at ohm.uwaterloo.ca

- listens to port 80 for Client Connection Requests
- Establish InputStream for sending data to client
- Establish OutputStream for receiving data from client

TCP connection example: (Server)

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class myserver {
    public static void main( String [] s) {
        try {
            ServerSocket s = new ServerSocket( 80 );
            While (true) {
                // wait for a connection request from client
                Socket clientConn = s.accept();
                InputStream in = clientConn.getInputStream();
                OutputStream out = clientConn.getOutputStream();
                // communicate with client
                // ..
                clientConn.close(); // close client connection
            }
        } catch (Exception e) {
            System.out.println("Exception!");
            // do something about the exception
        }
    }
}
```

```
TCP connection example: (Client)
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class myclient {
    public static void main( String [] s) {
        try {
            InetAddress addr = InetAddress.getByName(
                "ohm.uwaterloo.ca");
            Socket s = new Socket(addr, 80);
            InputStream in = s.getInputStream();
            OutputStream out = s.getOutputStream();
            // communicate with remote process
            // e.g. GET document /~ece454/index.html
            s.close();
        } catch(Exception e) {
            System.out.println("Exception");
            // do something about the Exception
        }
    }
}
```

3.7 DATAGRAMS

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

The UDP protocol provides a mode of network communication where packets sent by applications are called datagrams. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The datagram Packet and Datagram Socket classes in the java.net package implement system independent datagram communication using UDP.

Actually the DatagramPacket class is a wrapper for an array of bytes from which data will be sent or into which data will be received. It also contains the address and port to which the packet will be sent.

DatagramPacket constructors:

```
public DatagramPacket(byte[] data, int length)
public DatagramPacket(byte[] data, int length, InetAddress host, int port)
```

You can construct a DatagramPacket object by passing an array of bytes and the number of those bytes to the DatagramPacket() constructor:

```
String s = "My first UDP Packet"
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length());
```

Normally the object of DatagramPacket is created by passing in the host and port to which you want to send the packet with data and its length. For example, object m in the code given below:

```
try
{
    InetAddress m = new InetAddress("http://mail.yahoo.com");
    int chargen = 19;
```

```
String s = "My second UDP Packet"
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length, m, chargin);
}
catch (UnknownHostException ex)
{
System.err.println(ex);
}
```

The byte array that's passed to the constructor is stored by reference, not by value. If you change its contents elsewhere, the contents of the `DatagramPacket` change as well.

`DatagramPackets` themselves are not immutable. You can change the data, the length of the data, the port, or the address at any time using the following four methods:

```
public void setAddress(InetAddress host)
public void setPort(int port)
public void setData(byte buffer[])
public void setLength(int length)
```

You can retrieve address, port, data , and length of data using the following four get methods:

```
public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()
```

`DatagramSocket` constructors: The `java.net.DatagramSocket` class has three constructors:

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

The first is used for datagram sockets that are primarily intended to act as clients, i.e., a sockets that will send datagrams before receiving anything from anywhere.

The second constructors that specify the port and optionally the IP address of the socket, are primarily intended for servers that must run on a well-known port.

Sending UDP Datagrams

To send data to a particular server, you first must convert the data into byte array. Next you pass this byte array, the length of the data in the array (most of the time this will be the length of the array) the `InetAddress` and port to `DatagramPacket()` constructor.

For example, first you create `atagramPacket` object

```
try
{
InetAddress m = new InetAddress("http://mail.yahoo.com");
int chargin = 19;
String s = "My second UDP Packet";
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length, m, chargin);
}
catch (UnknownHostException ex)
{
System.err.println(ex);
}
```

```
}
```

Now create a DatagramSocket object and pass the packet to its send() method as try

```
{  
    DatagramSocket sender = new DatagramSocket();  
    sender.send(dp);  
}  
catch (IOException ex)  
{  
    System.err.println(ex);  
}
```

Receiving UDP Datagrams

To receive data sent to you, construct a DatagramSocket object bound to the port on which you want to receive the data. Then you pass an empty DatagramPacket object to the DatagramSocket's receive() method.

public void receive(DatagramPacket dp) throws IOException.

The calling threads blocks until the datagram is received. Then dp is filled with the data from that datagram. You can then use getPort() and getAddress() to tell where the packet came from, getData() to retrieve the data, and getLength() to see how many bytes were in the data. If the received packet is too long for the buffer, then it is truncated to the length of the buffer. You can write program with the help of code written below:

```
try  
{  
    byte buffer = new byte[65536]; // maximum size of an IP packet  
    DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);  
    DatagramSocket ds = new DatagramSocket(2134);  
    ds.receive(dp);  
    byte[] data = dp.getData();  
    String s = new String(data, 0, data.getLength());  
    System.out.println("Port " + dp.getPort() + " on " + dp.getAddress()  
        + " sent this message:");  
    System.out.println(s);  
}  
catch (IOException ex)  
{  
    System.err.println(ex);  
}
```

Check Your Progress 3

- 1) Write a program to print the address of your local machine and Internet web site rediff.com and webduniya.com.

.....

.....

.....

.....

- 2) Give a brief explanation of TCP Client/Server Interaction.

.....

.....

.....

- 3) Differentiate between TCP or UDP Protocols.

.....

.....

.....

- 4) Write a program, which does UDP Port scanner by checking out various port numbers status (i.e.) Are they occupied or free?

.....

.....

.....

.....

.....

3.8 SUMMARY

You have learnt that Java provides stream sockets and datagram sockets. With stream sockets a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams.

Stream sockets provide connection-oriented service. The TCP protocol is used for this purpose. With datagram sockets individual packets of information are transmitted. UDP protocol is used for this kind of communication. Stream based connections are managed with Sockets objects.

Datagram packets are used to create the packets to send and receive information using Datagram Sockets. Connection oriented services can be seen as your telephone service and connection-less services can be seen as Radio Broadcast.

Domain naming services solve the problem of remembering the long IP address of various web sites and computers. You also have learn that there are some dedicated Port numbers for various protocols which are known as reserved port like for FTP you have port no. 21.

With Proxy servers you can prevent your computer not accessible to anyone you don't want. Your computer data cannot be traced easily if you are using proxy servers, as the IP addresses will not be known to the second person directly.

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1)
Client side Programming
import java.io.*;
import java.net.*;

```
public class Client
{
    public static final int DEFAULT_PORT = 8000;
    public static void usage()
    {
        System.out.println("Usage: java Client <hostname> [<port>]");
        System.exit(0);
    }
}
```

```
}
public static void main(String[] args)
{
    int port = DEFAULT_PORT;
    Socket s = null;

    // Parse the port specification
    if ((args.length != 1) && (args.length != 2)) usage();
    if (args.length == 1) port = DEFAULT_PORT;
    else
    {
        try
        {
            port = Integer.parseInt(args[1]);
        }
        catch (NumberFormatException e)
        { usage();
        }
    }
    try
    {
        // Here is a socket to communicate to the specified host and port
        s = new Socket(args[0], port);

        BufferedReader sin = new BufferedReader(new
        InputStreamReader(s.getInputStream())); //stream for reading
        PrintStream sout = new PrintStream(s.getOutputStream());
            // stream for writing lines of text
        // Here is a stream for reading lines of text from the console
        BufferedReader in = new BufferedReader(new
        InputStreamReader(System.in));

        System.out.println("Connected to " + s.getInetAddress()
            + ":" + s.getPort());
        String line;
        while(true)
        {
            // print a prompt
            System.out.print("> ");
            System.out.flush();
            // read a line from the console; check for EOF
            line = in.readLine();
            if (line == null) break;
            // Send it to the server
            sout.println(line);
            // Read a line from the server.
            line = sin.readLine();
            // Check if connection is closed (i.e. for EOF)
            if (line == null)
            {
                System.out.println("Connection closed by server.");
                break;
            }
            // And write the line to the console.
            System.out.println(line);
        }
    }
    catch (IOException e)
```



```

    {
        System.err.println(e);
    }
    finally
    {
        try
        {
            if (s != null)
                s.close();
        }
        catch (IOException e2) { ; }
    }
}
}

```

Server Side Programming

```

import java.io.*;
import java.net.*;
public class Server extends Thread
{
    public final static int DEFAULT_PORT = 8000;
    protected int port;
    protected ServerSocket listen_socket;
    public static void fail(Exception e, String msg)
    {
        System.err.println(msg + ": " + e);
        System.exit(1);
    }
    // Creating a ServerSocket to listen for connections on;
    public Server(int port)
    {
        if (port == 0) port = DEFAULT_PORT;
        this.port = port;
        try { listen_socket = new ServerSocket(port);
        }
        catch (IOException e)
        {
            fail(e, "Exception creating server socket");
        }
        System.out.println("Server: listening on port " + port);
        this.start();
    }
    // create a Connection object to handle communication through the new Socket.
    public void run()
    {
        try
        {
            while(true)
            {
                Socket client_socket = listen_socket.accept();
                Connection c = new Connection(client_socket);
            }
        }
        catch (IOException e)
        {
            fail(e, "Exception while listening for connections");
        }
    }
}

```

```
    }  
    }  
    public static void main(String[] args)  
    {  
        int port = 0;  
        if (args.length == 1)  
        {  
            try  
            {  
                port = Integer.parseInt(args[0]);  
            }  
            catch (NumberFormatException e)  
            {  
                port = 0;  
            }  
        }  
        new Server(port);  
    }  
}  
  
// A thread class that handles all communication with a client  
class Connection extends Thread  
{  
    protected Socket client;  
    protected BufferedReader in;  
    protected PrintStream out;  
    // Initialize the streams and start the thread  
    public Connection(Socket client_socket)  
    {  
        client = client_socket;  
        try  
        {  
            in = new BufferedReader(new InputStreamReader(client.getInputStream()));  
            out = new PrintStream(client.getOutputStream());  
        }  
        catch (IOException e)  
        {  
            try  
            {  
                client.close();  
            }  
            catch (IOException e2) { ; }  
            System.err.println("Exception while getting socket streams: " + e);  
            return;  
        }  
        this.start();  
    }  
    public void run()  
    {  
        String line;  
        StringBuffer revline;  
        int len;  
        try  
        {  
            for(;;)  
            {  
                // read in a line  
                line = in.readLine();  
            }  
        }  
        catch (IOException e)  
        {  
            return;  
        }  
    }  
}
```

```

        if (line == null) break;
        // reverse it
        len = line.length();
        revline = new StringBuffer(len);
        for(int i = len-1; i >= 0; i--)
            revline.insert(len-1-i, line.charAt(i));
        // and write out the reversed line
        out.println(revline);
    }
}
catch (IOException e) { ; }
finally
{
    try
    {
        client.close();
    }
    catch (IOException e2) {}
}
}
}

```

Output :

```

C:\WINNT\system32\cmd.exe - java Server
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
C:\Documents and Settings\Administrator>cd\
C:\>cd jdk1.3
C:\jdk1.3>cd bin
C:\jdk1.3\bin>java Server
Server: listening on port 8000

C:\WINNT\system32\cmd.exe - java Client Archana
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
C:\Documents and Settings\Administrator>cd\
C:\>cd jdk1.3
C:\jdk1.3>cd bin
C:\jdk1.3\bin>java Client Archana
Connected to Archana/10.10.13.215:8000
> Hello Good Morning
gnirroM dooG olleH

```

In the output screen you can see that that server is running and listening to on port number 8000.

At the client side you can see that whenever you will write a string as “Hello Good Morning”

Then the string goes to server side and the server reverses it as the reverse function is written on the server.

2) In Table given below Socket type protocols and their description is given:

Table 1:Types of Sockets

Socket type	Protocol	Description
SOCK_STREAM	Transmission Control Protocol (TCP)	The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent.

SOCK_DGRAM	User Datagram Protocol (UDP)	The datagram socket (SOCK_DGRAM) interface defines a connectionless service for datagrams, or messages. Datagrams are sent as independent packets. The reliability is not guaranteed, data can be lost or duplicated, and datagrams can arrive out of order. However, datagram sockets have improved performance capability over stream sockets and are easier to use.
SOCK_RAW	IP, ICMP, RAW	The raw socket (SOCK_RAW) interface allows direct access to lower-layer protocols such as Internet Protocol (IP).

Note:

The type of socket you use is determined by the data you are transmitting:

- When you are transmitting data where the integrity of the data is high priority, you ***must*** use stream sockets.
 - When the data integrity is not of high priority (for example, for terminal inquiries), use datagram sockets because of their ease of use and higher performance capability.
- 3) There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

Datagram communication:

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, socket address is required each time a communication is made.

Stream communication:

The stream communication protocol is known as TCP (Transfer Control Protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Check Your Progress 2

1. Using an anonymous proxy server you don't give anybody a chance to find out your IP address to use it in their own interests. We can offer you two ways to solve your IP problem:
 - i) Secure Tunnel - a pay proxy server with plenty of features. Effective for personal use, when your Internet activities are not involved in very active surfing, web site development, mass form submitting, etc. In short, Secure Tunnel is the best solution for most of Internet users. Ultimate protection of privacy - nobody can find out where you are engaged in surfing. Blocks all methods of tracking. Provides an encrypted connection for all forms of web browsing, including http, news, mail, and the especially vulnerable IRC and ICQ. Comes with special totally preconfigured software.

- ii) ProxyWay - a proxy server agent which you use together with your web browser to ensure your anonymity when you surf the Internet. It contains a database of anonymous proxy servers and allows you to easily test their anonymity. Using a network of publicly accessible servers ProxyWay shields your current connection when you visit websites, download files, or use web-based e-mail accounts.
- 2) Our own small proxy list is also a good place to start with if you are a novice.

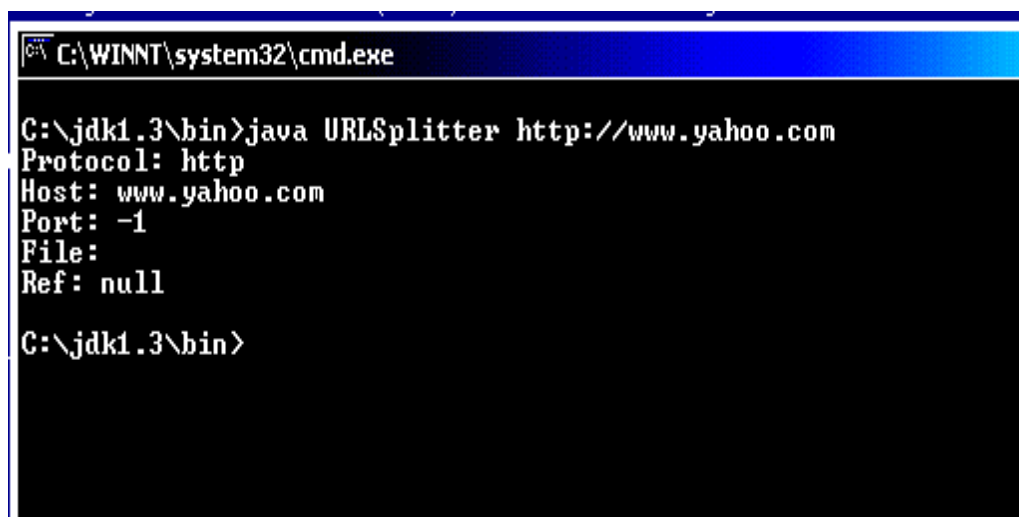
There are two types of Domain Name Servers: primary and secondary.

A primary server maintains a set of configuration files that contain information for the subset of the name space for which the server is authoritative. For example, the primary server for *indiatimes.com* contains IP addresses for all hosts in the *indiatimes.com* domain in configuration files. Resource Records are the entries in the configuration files that contain the actual data. A secondary server does not maintain any configuration files, but it copies the configuration files from the primary server in a process called a zone transfer. A secondary name server can respond to requests for name resolution, and it looks just like a primary name server from a user's perspective. Primary and secondary Domain Name Servers provide both performance and fault-tolerance benefits because you can split the workload between the servers, and if one goes down, the other can take over.

3)

```
import java.net.URL;
public class URLSplitter {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            try {
                java.net.URL u = new java.net.URL(args[i]);
                System.out.println("Protocol: " + u.getProtocol());
                System.out.println("Host: " + u.getHost());
                System.out.println("Port: " + u.getPort());
                System.out.println("File: " + u.getFile());
                System.out.println("Ref: " + u.getRef());
            }
            catch (java.net.MalformedURLException e) {
                System.err.println(args[i] + " is not a valid URL");
            }
        }
    }
}
```

Here's the output:



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

C:\jdk1.3\bin>java URLSplitter http://www.yahoo.com
Protocol: http
Host: www.yahoo.com
Port: -1
File:
Ref: null

C:\jdk1.3\bin>
```

Check Your Progress 3

Program to print the addresses:

```
1)
import java.net.*;
class I_Address
{
    public static void main(String[] args) throws UnknownHostException
    {
        InetAddress Addr = InetAddress.getLocalHost() ;
        System.out.println(Addr);
        Addr = InetAddress.getByName("yahoo.com");
        System.out.println(Addr);
        InetAddress Addr1[] = InetAddress.getAllByName("indiatimes.com") ;
        for( int i = 0; i < Addr1.length; i++)
            System.out.println(Addr1[i]);
    }
}
```

Output:

Run this program on your machine while connected to Internet to get proper output otherwise UnkwnHostException will occur.

Output on the machine is:

shashibhushan/190.10.19.205

rediff.com/208.184.138.70

webduniya.com/65.182.162.66

2) TCP Client/Server Interaction:

The Server starts by getting ready to receive client connections...

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
 - a. Accept new connection
 - b. Communicate
 - c. Close the connection

3) In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you receive are put in the order in which they were sent.

In short, TCP is useful for implementing network services-such as remote login (rlogin, telnet) and file transfer (FTP)- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in

implementing client/server applications in distributed systems built over local area networks.

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

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

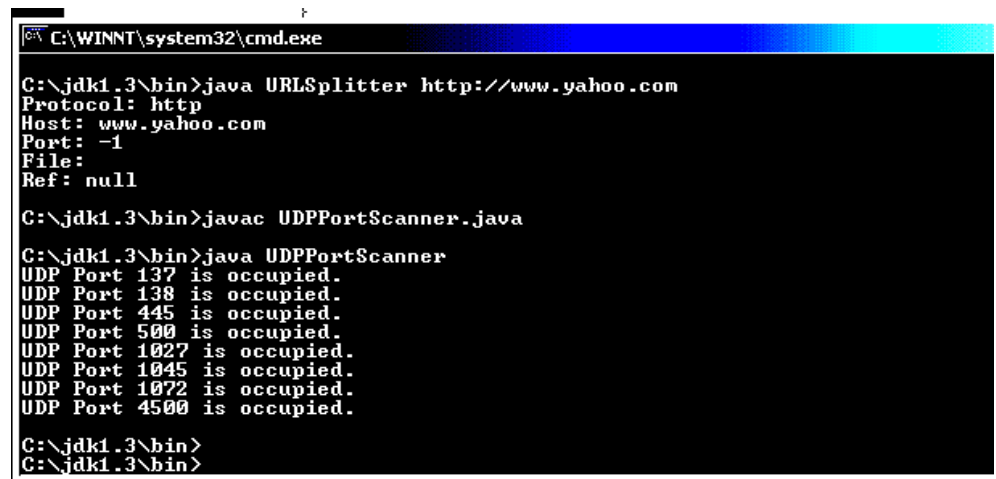
- 4) The LocalPortScanner developed earlier only found TCP ports. The following program detects UDP ports in use. As with TCP ports, you must be root on Unix systems to bind to ports below 1024.

```
import java.net.*;
import java.io.IOException;
public class UDPPortScanner {
    public static void main(String[] args) {
        // first test to see whether or not we can bind to ports
        // below 1024
        boolean rootaccess = false;
        for (int port = 1; port < 1024; port += 50) {
            try {
                ServerSocket ss = new ServerSocket(port);
                // if successful
                rootaccess = true;
                ss.close();
                break;
            }
            catch (IOException ex) {
            }
        }
        int startport = 1;

        if (!rootaccess) startport = 1024;
        int stopport = 65535;

        for (int port = startport; port <= stopport; port++)
        {
            try {
                DatagramSocket ds = new DatagramSocket(port);
                ds.close();
            }
            catch (IOException ex) {
                System.out.println("UDP Port " + port + " is occupied.");
            }
        }
    }
}
```

Output



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

C:\jdk1.3\bin>java URLSplitter http://www.yahoo.com
Protocol: http
Host: www.yahoo.com
Port: -1
File:
Ref: null

C:\jdk1.3\bin>javac UDPPortScanner.java

C:\jdk1.3\bin>java UDPPortScanner
UDP Port 137 is occupied.
UDP Port 138 is occupied.
UDP Port 445 is occupied.
UDP Port 500 is occupied.
UDP Port 1027 is occupied.
UDP Port 1045 is occupied.
UDP Port 1072 is occupied.
UDP Port 4500 is occupied.

C:\jdk1.3\bin>
C:\jdk1.3\bin>
```

Since UDP is connectionless it is not possible to write a remote UDP port scanner. The only way you know whether or not a UDP server is listening on a remote port is if it sends something back to you.