Chapter Four

Networking in Java

3.1.Introduction

The Internet is a global network of millions of computers. Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN). When a computer needs to communicate with another computer, it needs to know the other computer's address. An Internet Protocol (IP) address uniquely identifies the computer on the Internet. An IP address consists of four dotted decimal numbers between 0 and 255. Since it is not easy to remember so many numbers, they are often mapped to meaningful names called domain names. Special servers called Domain Name Servers (DNS) on the Internet translate host names into IP addresses. When a computer contacts a domain name, it first asks the DNS to translate this domain name into a numeric IP address and then sends the request using the IP address. The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets. Two higher-level protocols used in conjunction with the IP are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. UDP is a standard, low-overhead, connectionless, host-to-host protocol that is used over the IP. UDP allows an application program on one computer to send a data-gram to an application program on another computer.

Java supports both stream-based and packet-based communications. Stream-based communications use TCP for data transmission, whereas packet-based (Datagram-based) communications use UDP. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission.

At the core of Java's networking support is the concept of a socket. A socket identifies an endpoint in a network. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a port, which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O. Networking is tightly integrated in Java. The Java API provides the classes for creating sockets to facilitate program communications over the Internet. Sockets are the endpoints of logical connections between two hosts and can be used to send and receive data. Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files. Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets. The server must be running when a client attempts to connect to the server. The server waits for a connection request from a client. Network program is composed of two sockets; a server socket and client socket.

3.2.Server Sockets

To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services. For instance, the email server runs on port 25, and the Web server usually runs on port 80. You can choose any port number that is not currently used by other programs. The following statement creates a server socket serverSocket:

```
ServerSocket serverSocket = new ServerSocket(port);
```

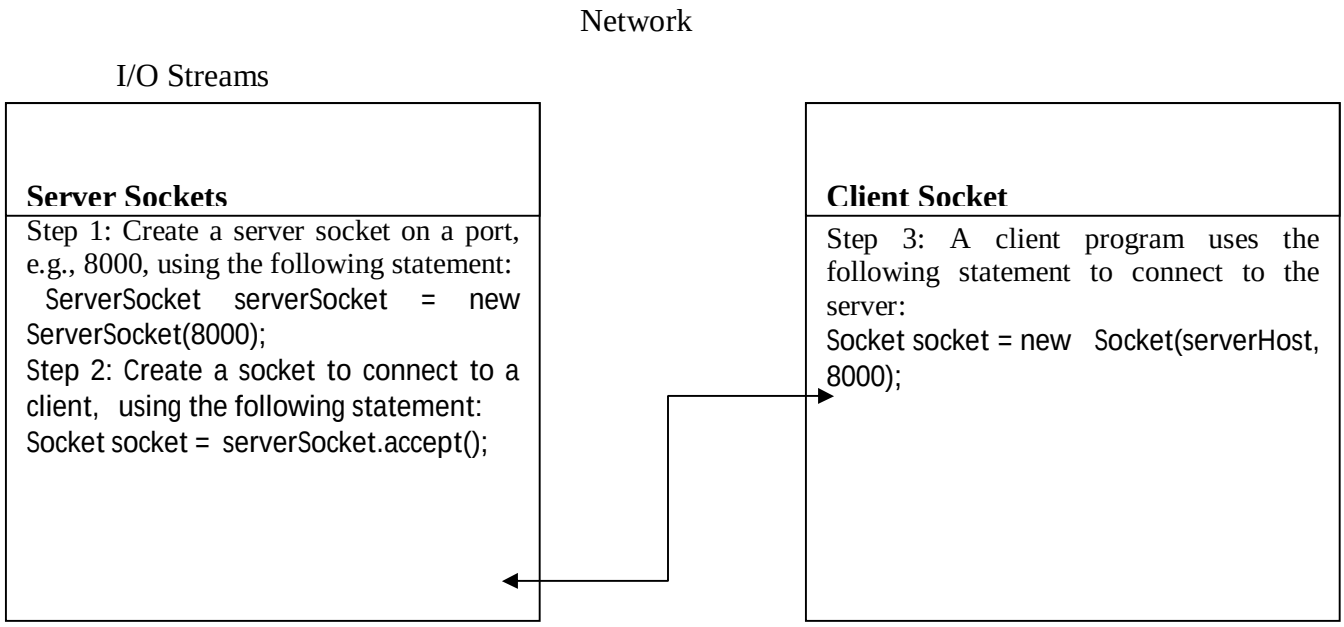

Figure 4.1. The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

After a server socket is created, the server can use the following statement to listen for connections: Socket socket = serverSocket.accept();

3.3.Client Socket

This statement waits until a client connects to the server socket. The client issues the following statement to request a connection to a server: Socket socket = new Socket(serverName, port); This statement opens a socket so that the client program can communicate with the server. serverName is the server's Internet host name or IP address. The following statement creates a socket on the client machine to connect to the host 172.10.144.53 at port 8000:

Socket socket = new Socket ("172.10.144.53", 8000)

Alternatively, you can use the domain name to create a socket, as follows:

Socket socket = new Socket ("www.facebook.com", 8000);

When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

3.4.The Networking Classes and Interfaces

Java supports TCP/IP both by extending the already established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the java.net package are shown here:

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
contentHandler	InterfaceAddress	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager	MulticastSocket	StandardSocketOption
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

The java.net package's interfaces are listed here:

ContentHandlerFactory	CookieStore	ProtocolFamily
CookiePolicy	DatagramSocketImplFactory	SocketImplFactory
	FileNameMap	SocketOption
		URLStreamHandler

3.5.Data Transmission through Sockets

After the server accepts the connection, communication between the server and client is conducted the same as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in the following Figure.

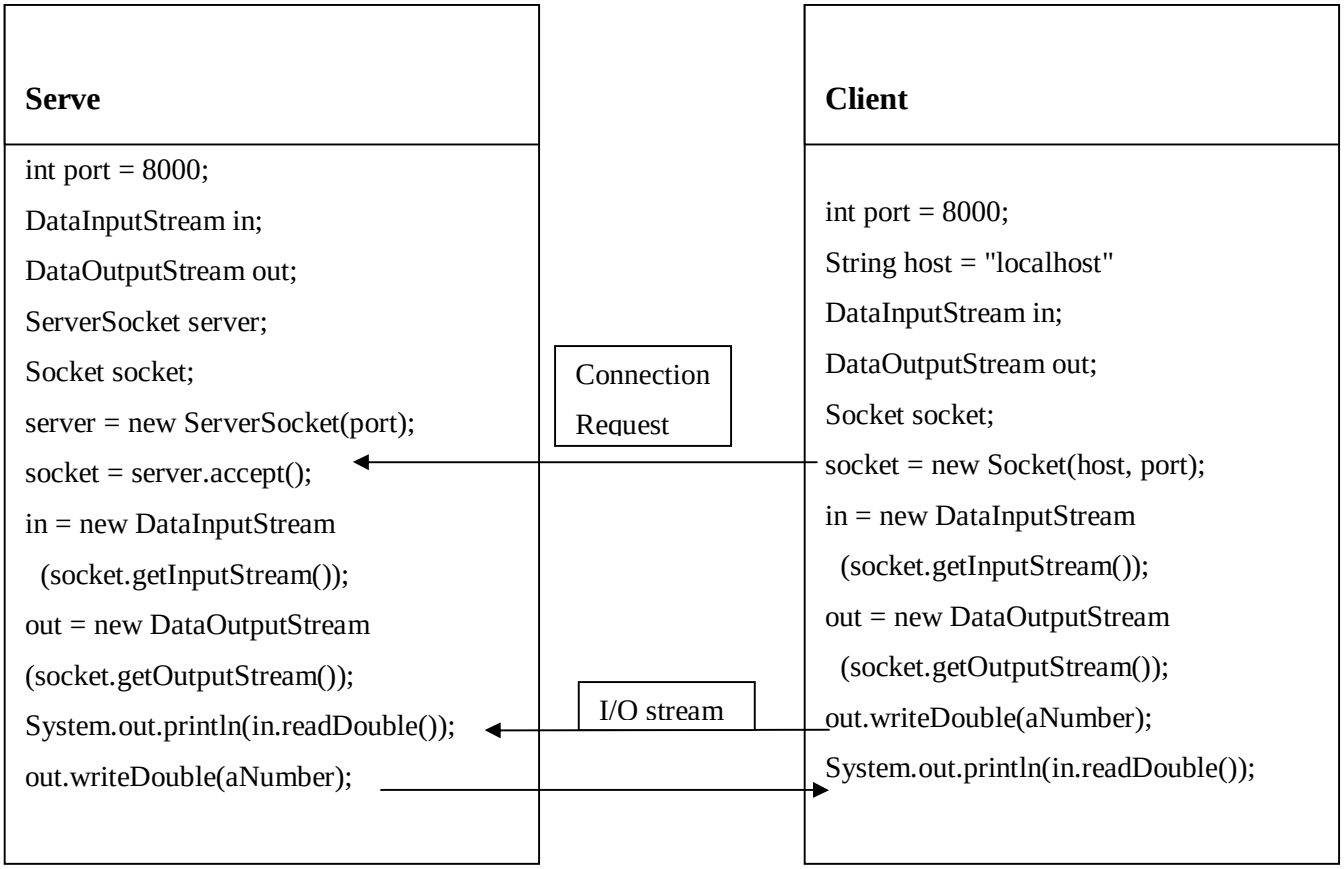

FIGURE 4.2 the server and client exchange data through I/O streams on top of the socket.

To get an input stream and an output stream, use the getInputStream () and getOutputStream() methods on a socket object. For example, the following statements create an InputStream stream called input and an OutputStream stream called output from a socket:

InputStream input = socket.getInputStream();

OutputStream output = socket.getOutputStream();

The InputStream and OutputStream streams are used to read or write bytes. You can use DataInputStream, DataOutputStream, BufferedReader, and PrintWriter to wrap on the InputStream and OutputStream to read or write data, such as int, double, or String. The following statements, for instance, create the DataInputStream stream input and the DataOutputStream output to read and write primitive data values:

```
DataInputStream input = new DataInputStream (socket.getInputStream());
DataOutputStream output = new DataOutputStream(socket.getOutputStream());
```

The server can use input.readDouble() to receive a double value from the client, and output.writeDouble(d) to send the double value d to the client.

A Client/Server Example

The following example presents a client program and a server program. The client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client comprise the radius of a circle, and the result produced by the server is the area of the circle.

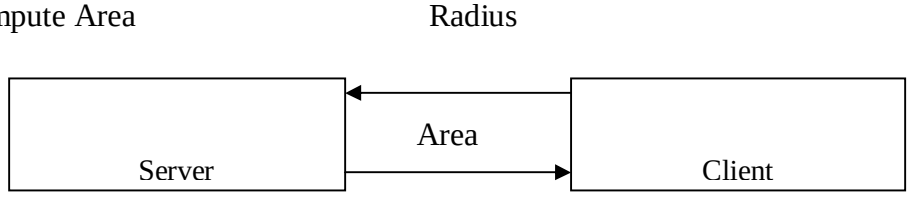

FIGURE 4.3 the client sends radius to the server; the server computes the area and sends it to the client.

The client sends the radius through a DataOutputStream on the output stream socket, and the server receives the radius through the DataInputStream on the input stream socket, as shown in. The server computes the area and sends it to the client through a DataOutputStream on the output

stream socket, and the client receives the area through a DataInputStream on the input stream socket.

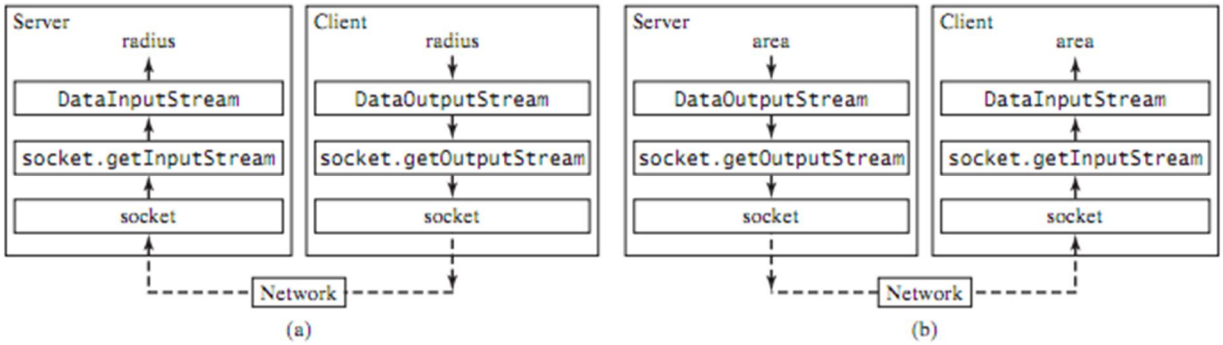

Figure 4.4 (a) the client sends the radius to the server. (b) The server sends the area to the client.

Server.java

```
import java.io.*;
import java.net.*;
import java.util.*;
public class Server {
public static void main(String[] args) {
new Server();
   }
public Server() {
try {
ServerSocket serverSocket = new ServerSocket(8000);
System.out.println("Server started at " + new Date());
// Listen for a connection request
Socket socket = serverSocket.accept();
DataInputStream inputFromClient = new
DataInputStream(socket.getInputStream());
DataOutputStream outputToClient = new
DataOutputStream(socket.getOutputStream());
while (true) {
double radius = inputFromClient.readDouble();
double area = radius * radius * Math.PI;
outputToClient.writeDouble(area);
System.out.println("Radius received from client: " + radius + '\n');
System.out.println("Area found: " + area + '\n');     }     }
catch(IOException ex) {      System.err.println(ex);  }     }     }
```

```
Client.java (Client side program)
import java.io.*;
import java.net.*;
import java.util.*;
public class Client {
// Text field for receiving radius
private DataOutputStream toServer;
private DataInputStream fromServer;
public static void main(String[] args) {
new Client();
   }
public Client() {
try {
Socket socket = new Socket("localhost", 8000);
fromServer = new DataInputStream(socket.getInputStream());
toServer = new DataOutputStream(socket.getOutputStream());
}
catch (IOException ex) { System.out.println(ex.toString);   }     }
double radius = Double.parseDouble(jtf.getText().trim());
toServer.writeDouble(radius);
toServer.flush();
double area = fromServer.readDouble();
System.out.println("Radius is " + radius + "\n");
System.out.println ("Area received from the server is" + area + '\n');  }
catch (IOException ex) { System.err.println(ex); }    }     }     }
```

You start the server program first, and then start the client program. In the client program, enter a radius in the text field and press Enter to send the radius to the server. The server computes the area and sends it back to the client. This process is repeated until one of the two programs terminates.

The networking classes are in the package java.net. You should import this package when writing Java network programs. The Server class creates a ServerSocket serverSocket and attaches it to port 8000, using this statement.

ServerSocket serverSocket = new ServerSocket(8000);

The server then starts to listen for connection requests, using the following:

Socket socket = serverSocket.accept();

The server waits until a client requests a connection. After it is connected, the server reads the radius from the client through an input stream, computes the area, and sends the result to the client through an output stream. The Client class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000:

Socket socket = new Socket("localhost", 8000);

If you run the server and the client on different machines, replace localhost with the server machine's host name or IP address. In this example, the server and the client are running on the same machine. If the server is not running, the client program terminates with a java.net.ConnectException. After it is connected, the client gets input and output streams wrapped by data input and output streams in order to receive and send data to the server. If you receive a java.net.BindException when you start the server, the server port is currently in use. You need to terminate the process that is using the server port and then restart the server.

Sending and Receiving Objects

Not only data of primitive types, we can also send and receive objects using ObjectOutputStream and ObjectInputStream on socket streams. To enable passing, the objects must be serializable.

In Java, serialization can be used for 2 things:

- Remote Method Invocation (RMI) - communication between objects via sockets
- Lightweight persistence - the archival of an object for use in a later invocation of the same program

Java provides two objects in java.io package

- ObjectInputStream
- ObjectOutputStream

Object serialization can take place over sockets as well as over a file in the same manner. An object is serializable only if its class implements the Serializable interface of java.io package.

The following example demonstrates how to send and receive objects. The example consists of three classes: Student.java, StudentClient.java, and StudentServer.java. The client program collects student information from a client and sends it to a server.

The Student class contains the student information: name and id of the student. The Student class implements the Serializable interface. Therefore, a Student object can be sent and received using the object output and input streams.

```
import java.io.Serializable;
public class Student implements Serializable {

    //private static final long serialVersionUID =
950169519310163575L;
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }


    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return
false;

        Student student = (Student) o;
```

```
            if (id != student.id) return false;
            if (name != null ? !name.equals(student.name) :
    student.name != null) return false;

            return true;
        }


        public int hashCode() {
            return id;
        }

        public String toString() {
            return "Id = " + getId() + " ; Name = " + getName();
        }       }
```

The client sends a Student object through an ObjectOutputStream on the out-put stream socket, and the server receives the Student object through the ObjectInputStream on the input stream socket. The client uses the writeObject method in the ObjectOutputStream class to send data about a student to the server, and the server receives the student's information using the readObject method in the ObjectInputStream class.

//StudentClient.java

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.net.SocketException;
public class Client {
    private Socket socket = null;
    private ObjectInputStream inputStream = null;
    private ObjectOutputStream outputStream = null;
    private boolean isConnected = false;
    public Client() {
    }
    public void communicate() {
        while (!isConnected) {
            try {
                socket = new Socket("localHost", 4445);
```

```
                System.out.println("Connected");
                isConnected = true;
                outputStream = new
ObjectOutputStream(socket.getOutputStream());
                Student student = new Student(1, "Anbesaw ");
                System.out.println("Object to be written = " + student);
                outputStream.writeObject(student);
            } catch (SocketException se) {
                se.printStackTrace();
                // System.exit(0);
            } catch (IOException e) {
                e.printStackTrace();
            }         }     }

    public static void main(String[] args) {
        Client client = new Client();
        client.communicate();
    }                        }
```

//Student server program

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
public class Server {
    private ServerSocket serverSocket = null;
    private Socket socket = null;
    private ObjectInputStream inStream = null;
    public Server() {
    }
    public void communicate() {
        try {
            serverSocket = new ServerSocket(4445);
            socket = serverSocket.accept();
            System.out.println("Connected");
            inStream = new ObjectInputStream(socket.getInputStream());
            Student student = (Student) inStream.readObject();
            System.out.println("Object received = " + student);
```

```
            socket.close();

        } catch (SocketException se) {
            System.exit(0);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException cn) {
            cn.printStackTrace();
        }    }
    public static void main(String[] args) {
        Server server = new Server();
        server.communicate();
    }     }
```

On the server side, when a client connects to the server, the server creates an ObjectInputStream on the input stream of the socket, invokes the readObject method to receive the Student object through the object input stream, and writes the object to a file.

Java Network Programming Using UDP

As discussed before , the transport layer has two protocols: TCP and UDP. We can use either TCP or UDP in our application. Java provides options to select either TCP or UDP in applications. This section describes Java Socket programming using UDP with example.

A comparison between TCP and UDP is given below as a table.

SL.No:	**TCP**	**UDP**
1	**TCP is connection oriented**	**Connection less**
2	**Reliable**	**Un-reliable**
3	**TCP is slower when compared with UDP**	**Faster**
4	**TCP header size is 20 bytes**	**UDP header size is 8 bytes**
5	**Flow control is there**	**No flow control**

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

- java.net.DatagramPacket
- java.net.DatagramSocket

The DatagramPacket class represents a data packet intended for transmission using the User Datagram Protocol. Packets are containers for a small sequence of bytes, and include addressing information such as an IP address and a port.

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

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 a 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.

DatagramSocket also defines the close ( ) method, which closes the socket. Beginning with JDK 7, DatagramSocket implements AutoCloseable, which means that a DatagramSocket can be managed by a try-with-resources block.

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)

In this example we have two Java classes. One class act as client and the other is the server. Let us see the server code first.

//Server UDP program

```
import java.io.*;
import java.net.*;
public class UDPSocketServer {
    DatagramSocket socket = null;
```

```
    public UDPSocketServer() {
    }
    public void createAndListenSocket() {
        try {
            socket = new DatagramSocket(9876);
            byte[] incomingData = new byte[1024];
            while (true) {
                DatagramPacket incomingPacket = new DatagramPacket
(incomingData, incomingData.length);
                socket.receive(incomingPacket);
                String message = new String(incomingPacket.getData());
                System.out.println("Received message from client: " +
message);
                InetAddress IPAddress = incomingPacket.getAddress();
                int port = incomingPacket.getPort();
                String reply = "Thank you for the message";
                byte[] data = reply.getBytes();
                DatagramPacket replyPacket =
                        new DatagramPacket(data, data.length, IPAddress,
port);
                socket.send(replyPacket);
                Thread.sleep(2000);
                socket.close();
            }
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException i) {
            i.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        UDPSocketServer server = new UDPSocketServer();
        server.createAndListenSocket();
    }
}
```

```
//UDP client
import java.io.IOException;
import java.net.*;
public class UDPSocketClient {
    DatagramSocket Socket;
    public UDPSocketClient() {
    }
    public void createAndListenSocket() {
        try {
            Socket = new DatagramSocket();
            InetAddress IPAddress = InetAddress.getByName("localhost");
            byte[] incomingData = new byte[1024];
            String sentence = "This is a message from client";
            byte[] data = sentence.getBytes();
            DatagramPacket    sendPacket    =    new    DatagramPacket(data,
   data.length, IPAddress, 9876);
            Socket.send(sendPacket);
            System.out.println("Message sent from client");
            DatagramPacket          incomingPacket          =          new
   DatagramPacket(incomingData, incomingData.length);
            Socket.receive(incomingPacket);
            String response = new String(incomingPacket.getData());
            System.out.println("Response from server:" + response);
            Socket.close();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }    }
    public static void main(String[] args) {
        UDPSocketClient client = new UDPSocketClient();
        client.createAndListenSocket();                  }             }
```