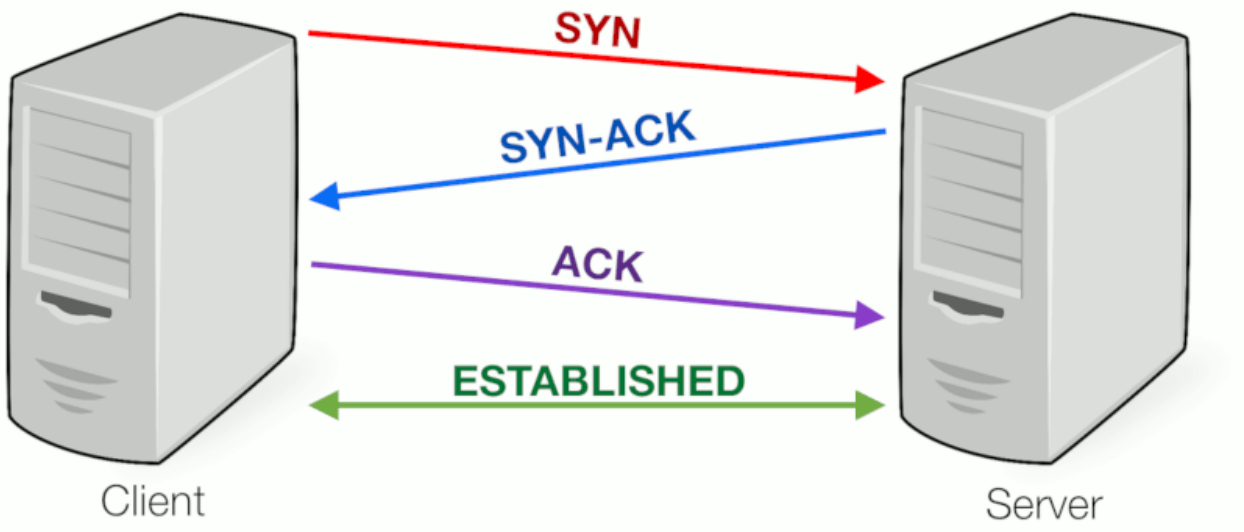
Unit - 5 Network Programming

Basics of networking

- Computer running on the internet communicate to each other using either TCP or UDP
- TCP
    - Connection oriented protocol
    - Provides reliable and ordered communication between devices
    - Establishes a connection before transferring data and ensures that data arrives intact and in correct order
    - Used for applications where accurate and complete delivery of data is crucial, such as web browsing (HTTP), file transfer(FTP), email(SMTP), telnet, etc.
- UDP
    - Connectionless protocol
    - Provides simple and lightweight communication without overhead of establishing and maintaining a connection
    - Does not guarantee delivery of order of packets, so often refused to as “unreliable” connection compared to TCP
    - Used in situations where real-time communication is important, and some data loss is acceptable, such as online gaming, streaming and video conferencing
- The choice between TCP and UDP depends on the specific requirements of the application

TCP

- It must first acknowledge the session
- Two computers must verify the connection before any communication takes place
- It does this by using 3 way handshake technique
- Working:
    - First, a sender computer will send a message called SYN
    - Receiving computer will send back an acknowledgement message telling the sender that it has received a message
    - And then the sender computer finally sends another acknowledgement message to the receiver
    - Finally, the connection is established and data is delivered



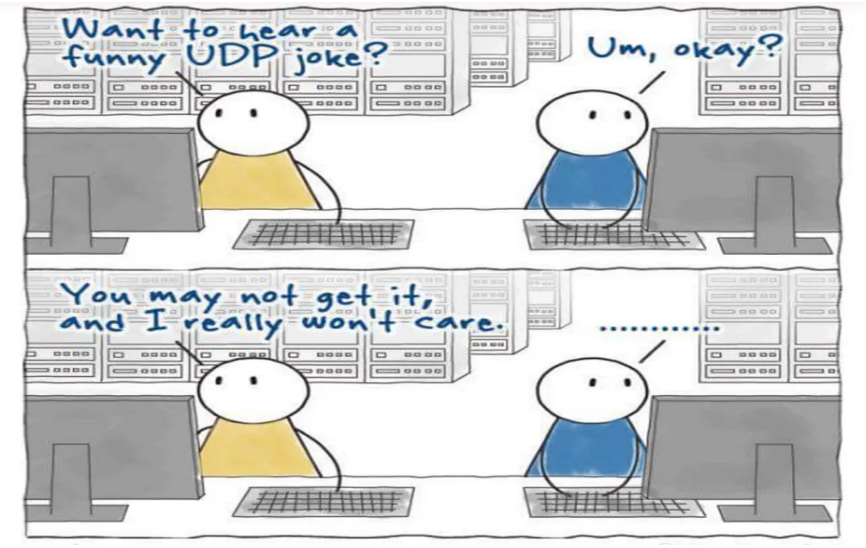
- TCP guarantees the delivery of data
- So if a packet goes astray, and it doesn't arrive, then TCP will resend it

Let me know if you are missing,
you any data that I send…

No problem, I will let
Know…

UDP

- It is also used for sending and receiving data
- But the difference is UDP is connectionless
- It doesn't establish session and doesn't guarantee data delivery
- When a computer sends a data, it doesn't really care if the data is received at the other end
- There is no guarantee in data delivery due to which there is less overhead and hence UDP is faster than TCP



Ports

- Ports play a crucial role in facilitating communication between processes on different devices
- It is not a physical connection. It's a logical connection that is used by programs and services to exchange information
- It specifically determines which program or service on a computer or server that is going to be used like pulling a web page, using FTP service, accessing email and so on.
- Ports have unique numbers that identify them that ranges from 0-65535 (2^{16} unsigned)
- Example:
    - 80 - HTTP
    - 443 - HTTPS
    - 21 - FTP
    - 25 - SMTP
- Port numbers are associated with IP addresses Ex. 192.168.1.15:80
- Example: when you want to connect to a server over an internet, the IP address is used to determine the geographical location of the server whereas, the port number determines which service or program on that server it wants to use
- Working with simple example(When we visit a web page, for example http://www.goole.com):
    - You open a browser and hit URL in the address bar
    - It converts domain name of Google.com into Google's IP address, i.e. 215.114.85.17
    - Since you are using HTTP, your computer adds port 80 to IP i.e. 215.114.85.17:80
    - Now IP is used to locate Google's server
    - Once location is found, IP will have done its job and IP will be discarded
    - Web server will now see incoming request with port number 80 and will forward that request to build in web service
    - Finally, Google's web page will be retrieved

- We can see current network connections and port activity on our computer using a command line tool called **netstat**
- Command : **netstat -a**
- Port number ranges from 0-65535
- These port numbers are assigned by IANA (Internet assigned number Authority)
- These port numbers are broken down into three categories:
    - **0-1023**
        - These port numbers are system or well-known ports
        - Used commonly everyday
        - Such as 80,443,21,etc.

    - **1024-49151**
        - These are called user or registered ports
        - Can be registered by companies and developers for a particular service
        - Ex: 1102 (Adobe server), 1527 (Oracle server)
    - **49152-65536**
        - These are called dynamic or private ports
        - These are client-side port that are free to use
        - These are the ports that our computer assigns temporarily to itself during a session
        - Ex: viewing a webpage

Networking classes in the JDK

- By using the package java.net, java programs can use TCP or UDP to communicate over the internet
- URL, URLConnection, Socket, ServerSocket classes all use TCP to communicate over the internet
- DatagramPacket, DatagramSocket, MulticastSocket classes are used for UDP

Working with URLs

- URL is the acronym for Uniform Resource Locator. It is a reference (an address) to a resource on the Internet.
- You provide URLs to your favorite Web browser so that it can locate files on the Internet, in the same way that you provide addresses on letters so that the post office can locate your correspondents.
- Java programs that interact with the Internet also may use URLs to find the resources on the Internet you wish to access.
- Java programs can use a class called URL in the java.net package to represent a URL address.
- The term URL can be ambiguous. It can refer to an Internet address or a URL object in a Java program.
- Here, "URL address" is used to mean an Internet address and "URL object" to refer to an instance of the URL class in a program.

URL

- URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet.
- If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.
- It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network.
- However, remember that URLs also can point to other resources on the network, such as database queries and comm and output.
  Example:
  http://localhost/phpmyadmin/index.php?route=/database/structure&db=college

- A URL has two main components:
    - Protocol identifier: For the URL http://example.com, the protocol identifier is http.
    - Resource name: For the URL http://example.com, the resource name is example.com.
- Note that the protocol identifier and the resource name are separated by a colon and two forward slashes.
- The protocol identifier indicates the name of the protocol to be used to fetch the resource.
- The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents.
- HTTP is just one of many different protocols used to access different types of resources on the internet. Other protocols include File Transfer Protocol (FTP), telnet, mailto, HTTPS, etc.
- The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:
    - **Host Name**
      The name of the machine on which the resource lives.
      Example: www.example.com
    - **Filename**
      The pathname to the file on the machine.
      Example: /path/to/resource/file.html
    - **Port Number**
      The port number to which to connect (typically optional).
      Example: :8080 (specifying port 8080)
    - **Reference**
      A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).
      Example: #section2

For many protocols, the host name and the filename are required, while the port number and reference are optional.

Complete URL :
http://www.example.com:8080/path/to/resource/file.html#section2

Creating a URL

- The easiest way to create a URL object is from a String that represents the human-readable form of the URL address.
- In your Java program, you can use a String containing this text to create a URL object:
  URL myURL = new URL("http://example.com/");
- The URL object created above represents an absolute URL.
- You can also create URL objects from a relative URL address.

Creating a Relative URL

- In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:
  http://example.com/pages/page1.html
  http://example.com/pages/page2.html
- You can create URL objects for these pages relative to their common base URL:
  http://example.com/pages/ like this:
  URL myURL = new URL("http://example.com/pages/");
  URL page1URL = new URL(myURL, "page1.html");
  URL page2URL = new URL(myURL, "page2.html");
- This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification.
- The general form of this constructor is:
  URL(URL baseURL, String relativeURL)
- The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base.

- If baseURL is null, then this constructor treats relativeURL like an absolute URL specification.
- Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.
- Other URL constructors:
  new URL("http", "example.com", "/pages/page1.html");
  - This is equivalent to
    new URL("http://example.com/pages/page1.html");
  - The first argument is the protocol, the second is the host name, and the last is the pathname of the file.
  - Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.
  - The final URL constructor adds the port number to the list of arguments used in the previous constructor:
    URL url = new URL("http", "example.com", 80, "pages/page1.html");
  - This creates a URL object for the following URL:
    http://example.com:80/pages/page1.html
  - If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method

URL addresses with special characters

- Some URL addresses contain special characters, for example the space character. Like this:
  http://example.com/hello world/
- To make these characters legal they need to be encoded before passing them to the URL constructor.
  URL url = new URL("http://example.com/hello%20world");
- Encoding the special character(s) in this example is easy as there is only one character that needs encoding,

- But for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the java.net.URI class to automatically take care of the encoding for you.
  URI uri = new URI("http", "example.com", "/hello world/", "");
- And then convert the URI to a URL.
  URL url = uri.toURL();

Malformed URL

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

```
try {
        URL myURL = new URL(...);
}
catch (MalformedURLException e) {
        // exception handler code here
        // ...
}
```

Parsing a URL

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

getProtocol
Returns the protocol identifier component of the URL.
Example : http
getAuthority
Returns the authority component of the URL.
Example : user:password@www.example.com:8080
getHost
Returns the host name component of the URL.
Example : www.example.com
getPort
Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
Example : 8080
getPath
Returns the path component of this URL.
Example : /path/to/resource/file.html
getQuery
Returns the query component of this URL.
Example : param1=value&param2=value
getFile
Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
Example : /path/to/resource/file.html?param1=value1&param2=value2
getRef
Returns the reference component of the URL.
Example : section2

- Note:
  Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs.
- The URL class is somewhat HTTP-centric.
- You can use these getXXX methods to get information about the URL regardless of the constructor that you used to create the URL object.

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

```java
import java.net.*;

class URLParse {
    public static void main(String[] args) {
        URL aURL;
        try {
            aURL = new URL("http://example.com:80/docs/books/tutorial"
                    + "/index.html?name=networking#DOWNLOADING");
            System.out.println("protocol = " + aURL.getProtocol());
            System.out.println("authority = " + aURL.getAuthority());
            System.out.println("host = " + aURL.getHost());
            System.out.println("port = " + aURL.getPort());
            System.out.println("path = " + aURL.getPath());
            System.out.println("query = " + aURL.getQuery());
            System.out.println("filename = " + aURL.getFile());
            System.out.println("ref = " + aURL.getRef());
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

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

Reading from URL

- After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL.
- The openStream() method returns a java.io.InputStream object, so reading from a URL is as easy as reading from an input stream.
- The following small Java program uses openStream() to get an input stream on the URL
  http://www.google.com.np/
- It then opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL.
- Everything read is copied to the standard output stream
- Example :

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;

public class ReadFromUrl {
    public static void main(String[] args) throws IOException {
        URL google;
        try {
            google = new URL("http://www.google.com.np/");
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(google.openStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null){
                System.out.println(inputLine);
            }
            in.close();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}

```

Connecting to a URL

- After you've successfully created a URL object, you can call the URL object's openConnection method to get a URLConnection object, or one of its protocol specific subclasses, e.g. java.net.HttpURLConnection
- You can use this URLConnection object to setup parameters and general request properties that you may need before connecting.
- Connection to the remote object represented by the URL is only initiated when the URLConnection.connect method is called.
- When you do this you are initializing a communication link between your Java program and the URL over the network.
- For example, the following code opens a connection to the site example.com:

```java
import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

public class URLConnectionDemo {
    public static void main(String[] args) {
        try {
            URL myURL = new URL("http://example.com/");
            URLConnection myURLConnection = myURL.openConnection();
            myURLConnection.connect();
        } catch (MalformedURLException e) {
            // new URL() failed
            // ...
        } catch (IOException e) {
            // openConnection() failed
            // ...
        }
    }
}

```

- A new URLConnection object is created every time by calling the openConnection method of the protocol handler for this URL.
- You are not always required to explicitly call the connect method to initiate the connection.

- Operations that depend on being connected, like getInputStream, getOutputStream, etc, will implicitly perform the connection, if necessary.
- Now that you've successfully connected to your URL, you can use the URLConnection object to perform actions such as reading from or writing to the connection. The next example shows how.

Reading from a URL connection

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

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;

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

```

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

Sockets

- URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet.
- Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.
- They offer a programming interface for network communication, allowing data to be sent and received between different applications or devices. **(two-way communication)**
- In client-server applications, the server provides some service, such as processing database queries or sending out current date and time
- The client uses the service provided by the server, either displaying database query results to the user or displaying the date and time to the user
- For this communication must be reliable which is provided by TCP (using point-to-point communication channel)
- To communicate over TCP:
    - a client program and a server program establish a connection to one another
    - Each program binds a socket to its end of the connection
    - To communicate, the client and the server each reads from and writes to the socket bound to the connection
- **Socket = IP address + port number**

What is socket?

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

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

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network.

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

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

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

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client.

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

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

Simple Server using ServerSocket class

Step 1: Create a ServerSocket

ServerSocket server = new ServerSocket(portNumber, queueLength);

- registers an available TCP port number and specifies themaximum number of clients that can wait to connect to the server (i.e., the queue length)
- The port number is used by clients to locate the server application on the server computer
- The constructor establishes the port where the server waits for connections from clients—a process known as **binding the server to the port**

Step 2: Wait for a Connection

Socket connection = server.accept();

- In Step 2, the server listens indefinitely (or blocks) for an attempt by a client to connect
- server.accept() returns a Socket when a connection with a client is established

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

- Step 3 is to get the OutputStream and InputStream objects that enable the server to communicate with the client by sending and receiving bytes
- The server invokes method getOutputStream on the Socket to get a reference to the Socket's OutputStream and invokes method getInputStream on the Socket to get a reference to the Socket's InputStream

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

PrintWriter out=new PrintWriter(con.getOutputStream(),true);

- whatever the client writes to its OutputStream (with a corresponding PrintWriter) is available via the server's InputStream

Step 4: Perform the Processing

- The server and the client communicate via the OutputStream and InputStream objects.

Step 5: Close the Connection

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

Simple Client using Socket class

Step 1: Create a Socket to Connect to the Server

- In first step we create a Socket to connect to the server. The Socket constructor establishes the connection

Socket connection = new Socket(serverAddress, port);

- Socket constructor with two arguments—the server's address (serverAddress) and the port number
- If the connection attempt is successful, this statement returns a Socket
- If it fails then it throws an instance of a subclass of IOException

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

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

Step 3: Perform the Processing

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

Step 4: Close the Connection

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

InetAddress class

- InetAddress class is used to convert between host names and Internet addresses
- The static getByName method returns an InetAddress object of a host

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

- It returns an InetAddress object that encapsulates the sequence of four bytes such as 132.163.4.104
- Sometimes host name corresponds to three different Internet addresses like java.sun.com
- You can get all hosts with the getAllByName method

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

Simple chat application performed between client and server

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

class ClientDemo{
    public static void main(String[] args) throws UnknownHostException, IOException {
        Socket socket = new Socket("localhost",4444);

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(),true);

        do{
            String messageFromServer = in.readLine();
            System.out.println(messageFromServer);

            BufferedReader terminalInput = new BufferedReader(new InputStreamReader(System.in));
            String messageFromClient = terminalInput.readLine();
            out.println("From client: "+messageFromClient);
            if(messageFromClient.equalsIgnoreCase("bye") || messageFromClient == null){
                break;
            }

        }while(true);


    }
}
```

A client program

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

public class ServerDemo {
    public static void main(String[] args) throws IOException{
        ServerSocket serverSocket = new ServerSocket(4444);
        System.out.println("Waiting for connection....");
        Socket socket =serverSocket.accept();
        System.out.println("Connected with: "+socket.getInetAddress());

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(),true);

        do{
            System.out.println("To client:");
            BufferedReader terminalInput = new BufferedReader(new InputStreamReader(System.in));
            String messageToClient = terminalInput.readLine();
            out.println("From server: "+messageToClient);

            String messageFromClient = in.readLine();
            if(messageFromClient.equalsIgnoreCase("bye") || messageFromClient == null){
                break;
            }
            System.out.println("From client: "+messageFromClient);
        }while(true);


        in.close();
        out.close();
        serverSocket.close();
    }
}

```

A server socket program

Serving Multiple Clients

- Clients from all over the internet may use server at the same time
- In normal client-server program as we have seen before, single client is connected for long time and other clients are rejected
- This can be improved using threads
- How?
    - Whenever accept() is success, we launch a new thread
    - New thread will take care of connection between client and server
    - Now main program will go back and wait for next connection
- Now the main program will look like this:

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

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

```
class ThreadedEchoHandler implements Runnable
{...
public void run()
{
        try
        {
                InputStream inStream = incoming.getInputStream();
                OutputStream outStream = incoming.getOutputStream();
                ...process input and send response...
```

```
                incoming.close();
            }
            catch(IOException e)
            {
            handle exception
            }
    }
    }
```

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

Socket Timeouts

- Used to limit the amount of time socket operation can take place
- To prevent from socket waiting indefinitely for connection, we can set the timeout

- For socket connection timeout:

  ```
  Socket socket = new Socket();
  SocketAddress address = new InetSocketAddress("example.com", 80);
  int timeout = 5000; // 5 seconds timeout
  socket.connect(address, timeout);
  ```

- For socket reading timeout:

  ```
  Socket socket = new Socket("example.com", 80);
  InputStream inputStream = socket.getInputStream();
  socket.setSoTimeout(5000); // 5 seconds timeout for read operations
  ```

- ServerSocket accept timeout:

```
ServerSocket serverSocket = new ServerSocket(8080);
int timeout = 5000; // 5 seconds timeout for accepting connections
serverSocket.setSoTimeout(timeout);

try {
    Socket clientSocket = serverSocket.accept();
    // Handle the accepted connection
} catch (SocketTimeoutException e) {

}
```

Interruptible Sockets

- When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed.
- Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.
- We cannot simply unblock it by calling the interrupt
- To interrupt a socket operation, you use a SocketChannel, a feature of the java.nio package:

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

- To read from a SocketChannel we can simply use Scanner class:

  OutputStream outStream = Channels.newOutputStream(channel);

- Similarly, to turn a channel into an output stream,we can use the static Channels.newOutputStream method.

OutputStream outStream = Channels.newOutputStream(channel);

- Whenever a thread is interrupted during an open, read, or write operation, the operation does not block but is terminated with an exception.

Half-Close

- A half-close in the context of socket communication refers to closing one direction of data flow while keeping the other open.
- In other words, one side of the socket (either the client or the server) stops sending data, but it can still receive data.
- This is often useful in scenarios where one party has finished sending data but wants to continue receiving responses.
- The client side looks like this:

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

Sending E-Mail via javax.mail API

- Sending emails using the javax.mail API in Java involves creating a Session and a Message object, configuring them with the necessary details, and then using a Transport to send the message.
- Below is a simple example demonstrating how to send an email using the javax.mail API.
- Note that you'll need to have the javax.mail library in your classpath.

Setting up java mail using mail API

Step 1 - Installing JavaMail API

First, you need to include two jar files into your CLASSPATH:

- **mail.jar**
- **Activation.jar**

After including jar files, you can start sending emails. However, you need an SMTP server to send emails using JavaMail API. You can easily set up an SMTP server using a provider like Mailtrap.

Step 2 - Getting the mail session

As the second step, you need to get the session object. The session object contains all the information related to the host like name, user name, password, etc. For that, you can write code like below.

```
// properties object contains host information
Properties properties = new Properties();
Session session=Session.getInstance(properties,null);
```

Also, you can pass the username and the password to obtain authentication for a network connection.

```
Session session = Session.getInstance(properties, new
javax.mail.Authenticator() {
  protected PasswordAuthentication getPasswordAuthentication() {
    return new PasswordAuthentication("sender@gmail.com", "***");
  }
});
```

Step 3 - Composing the email

You can use the javax.mail.internet.MimeMessage subclass to compose the message. First, you need to pass the session object to MimeMessage class. Then you can configure the sender, receiver, subject, and message body.

```
MimeMessage message = new MimeMessage(session);
// Sender email
message.setFrom(new InternetAddress(from));
// Receiver email
message.addRecipient(Message.RecipientType.TO, new
InternetAddress(to));
// Email subject
message.setSubject("This is the email subject");
// Email body
message.setText("This is the email body");
```

Step 4 - Sending the email

Finally, you can use the javax.mail.Transport class to send the email.

```
Transport.send(message);
```

The complete code example of sending an email using JavaMail APi will look like below:

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

public class SendMail {
    public static void main(String[] args) {
      String to = "receiver@gmail.com";
      String from = "sender@gmail.com";
      String host = "sandbox.smtp.mailtrap.io";

      Properties properties = System.getProperties();
      properties.put("mail.smtp.host", host);
      properties.put("mail.smtp.port", "2525");
      properties.put("mail.smtp.ssl.enable", "true");
      properties.put("mail.smtp.auth", "true");

      Session session = Session.getInstance(properties, new javax.mail.Authenticator(){
        protected PasswordAuthentication getPasswordAuthentication() {
          return new PasswordAuthentication("4103696ff47431", "f92313b104aa83");
        }
      });

      try {
        MimeMessage message = new MimeMessage(session);
        message.setFrom(new InternetAddress(from));
        message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));
        message.setSubject("This is the email subject");
        message.setText("This is the email body");

        Transport.send(message);
      } catch (MessagingException mex) {
        mex.printStackTrace();
      }
   }
}

```