COMPUTER NETWORK LAB: DAY 3

COMMUNICATION PROTOCOLS

- **Datagram Communication - UDP** (user datagram protocol), is a connectionless protocol- each time datagrams are send, the local socket descriptor and the receiving socket's address also need to be send.
- **Stream Communication - TCP** (transfer control protocol), connection-oriented protocol. Connection first to be established between 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.

SOCKET PROGRAMMING

- java.net.Socket class represents a socket
- java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.
- ServerSocket class has four constructors –
  - ✓ public ServerSocket(int port) throws IOException
  - ✓ public ServerSocket(int port, int backlog) throws IOException
  - ✓ public ServerSocket(int port, int backlog, InetAddress address) throws IOException
  - ✓ public ServerSocket() throws IOException

SOCKET PROGRAMMING

- common methods of the ServerSocket class –
- **public int getLocalPort()**
- **public Socket accept() throws IOException**
- **public void setSoTimeout(int timeout)**
- **public void bind(SocketAddress host, int backlog)**

SOCKET PROGRAMMING

- **java.net.Socket** class represents the socket that both the client and the server use to communicate with each other. The Socket class has five constructors that a client uses to connect to a server −
  - ✓ public Socket(String host, int port) throws UnknownHostException, IOException
  - ✓ public Socket(InetAddress host, int port) throws IOException
  - ✓ public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException
  - ✓ public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException
  - ✓ public Socket()

SOCKET PROGRAMMING

- Methods that can be invoked by both the client and the server:

  - ✓ public void connect(SocketAddress host, int timeout) throws IOException
  - ✓ public InetAddress getInetAddress()
  - ✓ public int getPort()
  - ✓ public int getLocalPort()
  - ✓ public SocketAddress getRemoteSocketAddress()
  - ✓ public InputStream getInputStream() throws IOException
  - ✓ public OutputStream getOutputStream() throws IOException
  - ✓ public void close() throws IOException

SOCKET PROGRAMMING

- InetAddress Class represents an Internet Protocol (IP) address. Methods of this class are:
  - static InetAddress getByAddress(byte[] addr)
  - static InetAddress getByAddress(String host, byte[] addr)
  - static InetAddress getByName(String host)
  - String getHostAddress()
  - String getHostName()
  - static InetAddress InetAddress getLocalHost()
  - String toString()

HOW TO OPEN A SOCKET?

When programming a client:

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

HOW TO OPEN A SOCKET?

When programming a server:

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

HOW TO OPEN A SOCKET?

When implementing a server also create a socket object from the ServerSocket in order to listen for and accept connections from clients:

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

HOW TO CREATE AN INPUT STREAM?

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);
}
```

HOW TO CREATE AN INPUT STREAM?

- Class DataInputStream allows 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. Depending on the type of data to be received from the server, one can use the appropriate function.

HOW TO CREATE AN INPUT STREAM?

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

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

HOW TO CREATE AN OUTPUT STREAM?

On the client side, 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);
}
```

HOW TO CREATE AN OUTPUT STREAM?

The class **PrintStream** has methods for displaying textual representation of Java primitive data types. Its **Write** and **println** methods are important here. Also, you may want to use the **DataOutputStream**:

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

HOW TO CREATE AN OUTPUT STREAM?

- 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.

HOW TO CREATE AN OUTPUT STREAM?

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);
}
```

Note: You can use the class DataOutputStream as mentioned earlier.

HOW TO CLOSE SOCKETS?

The output and input stream should always be closed before closing the socket. On the client side:

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

HOW TO CLOSE SOCKETS?

- On the server side:

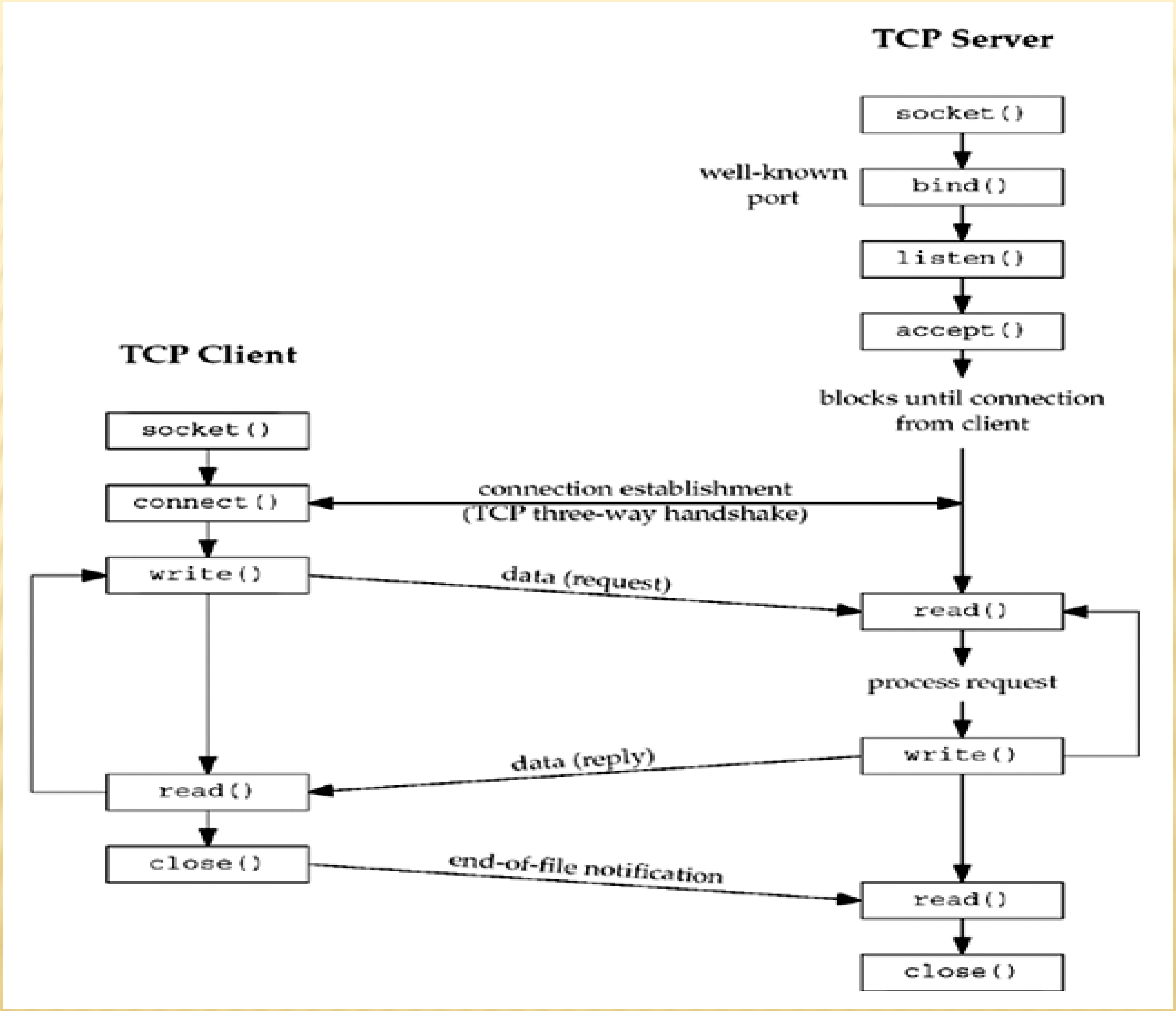
```
try {
        output.close();
        input.close();
        serviceSocket.close();
        MyService.close();
    } catch (IOException e)
    {
    System.out.println(e);
    }
```

IMPORTANT POINTS

- Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
- Then Server makes a new Socket to communicate with the client.

socket = server.accept()

- The accept() method blocks(just sits there) until a client connects to the server.
- Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.
- After we’re done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.


TCP Server
socket()
well-known port
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
connection establishment (TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

CONCLUSION

- When programming a client, you must follow these four steps:
    - Open a socket.
    - Open an input and output stream to the socket.
    - Read from and write to the socket according to the server's protocol.
    - Clean up.
- Only step that varies is step three, since it depends on the server you are talking to.

LAB ASSIGNMENT

- Design a prototype for implementing a program to obtain the IP address of the local or remote machine.