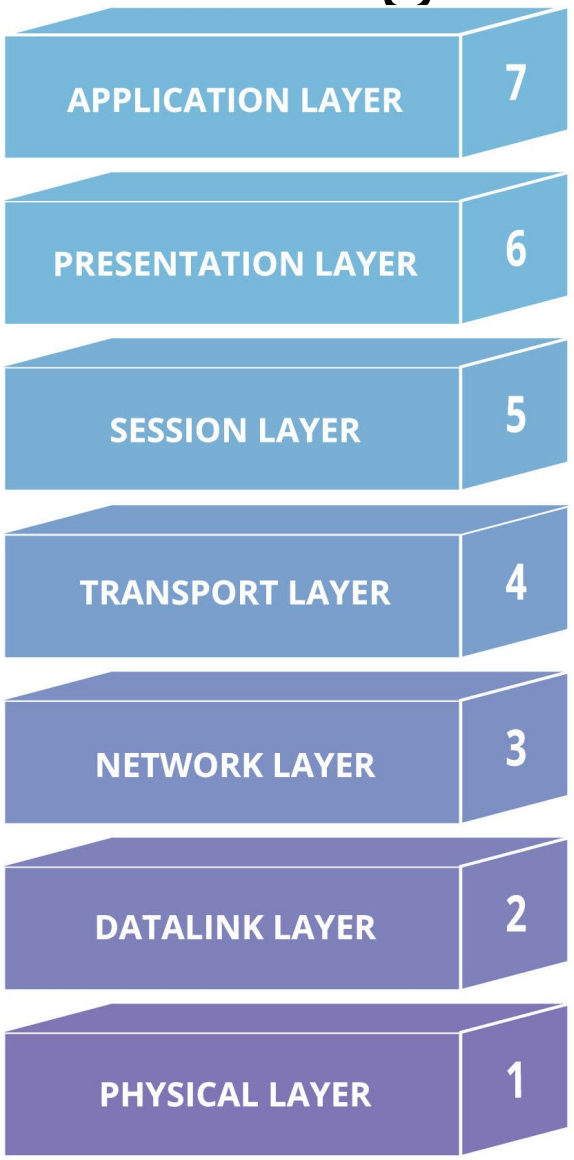
Day 4

Networking Model



- Human-computer interaction layer, where applications can access the network services
- Ensures that data is in a usable format and is where data encryption occurs
- Maintains connections and is responsible for controlling ports and sessions
- Transmits data using transmission protocols including TCP and UDP
- Decides which physical path the data will take
- Defines the format of data on the network
- Transmits raw bit stream over the physical medium

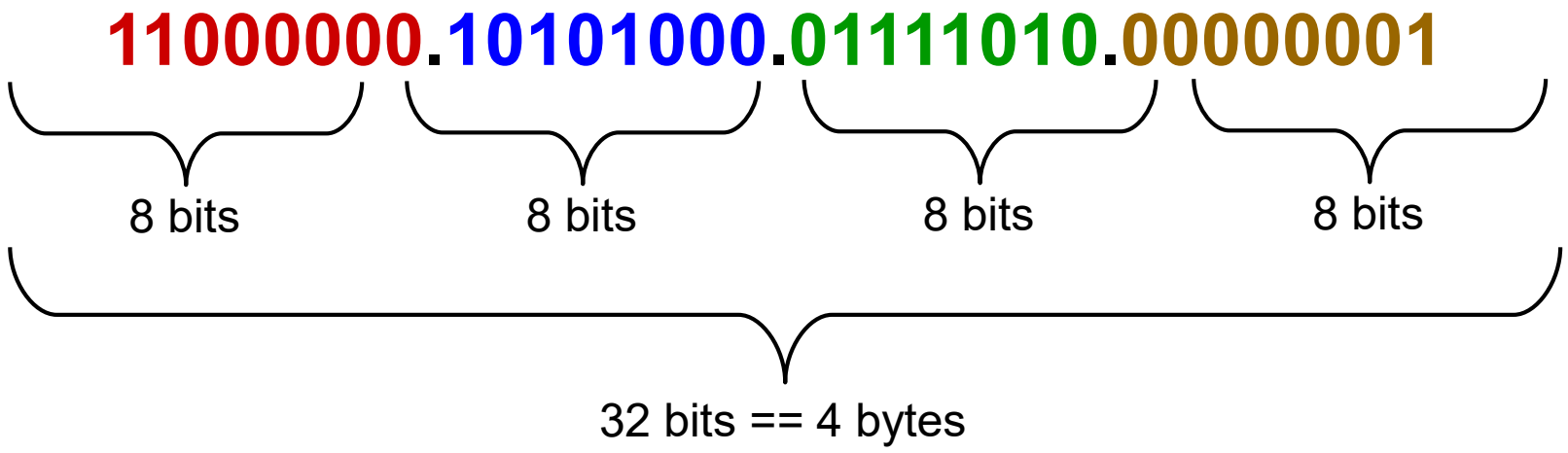
Image from https://www.cloudflare.com/en-gb/learning/ddos/glossary/open-systems-interconnection-model-osi/

HDB - Blocks and Units

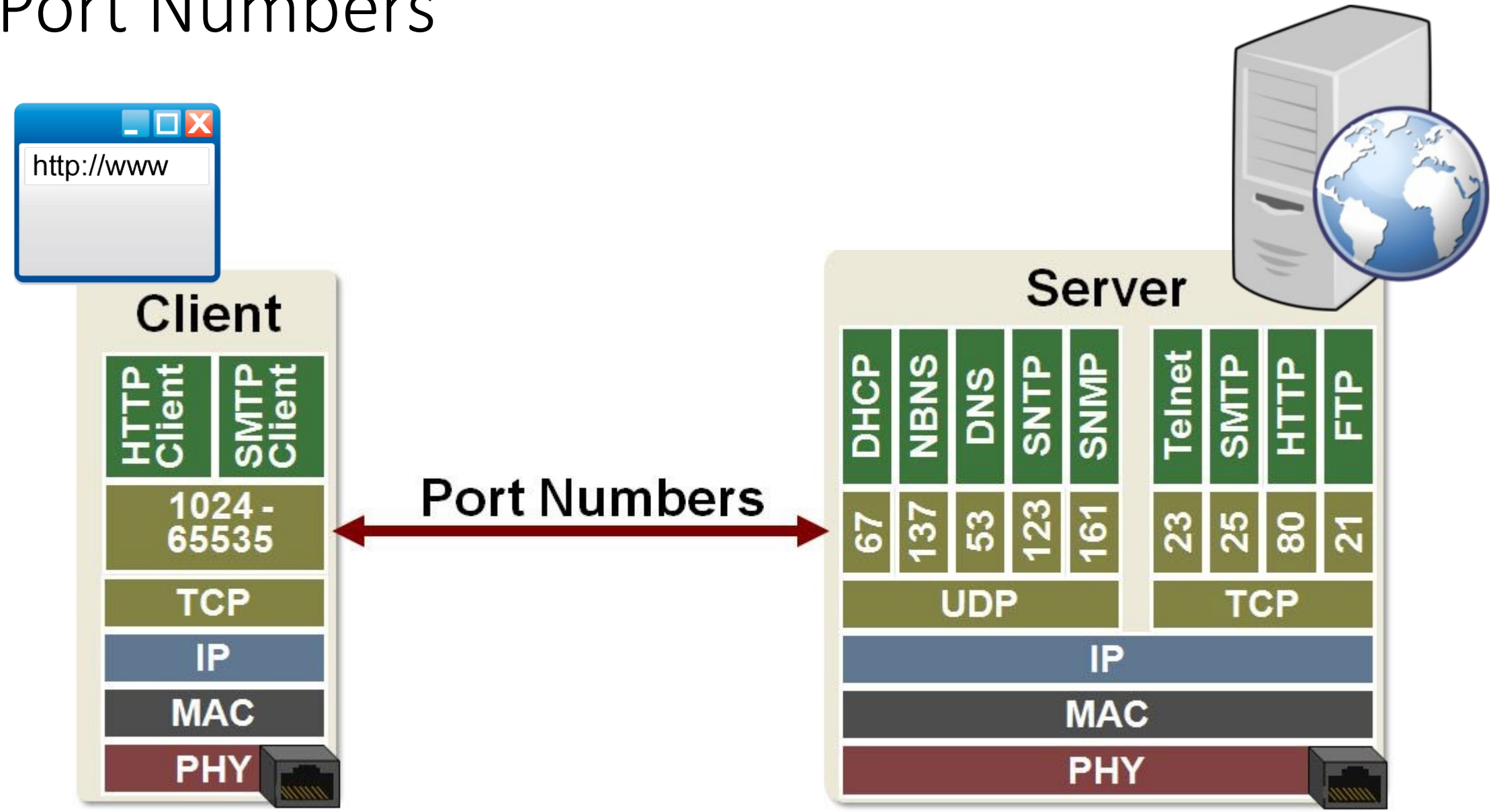
IP Addresses

- Unique identifiers assigned to any device that connects to the Internet
  - Usually assigned by the network provider that you connect to

192 . 168 . 122 . 1



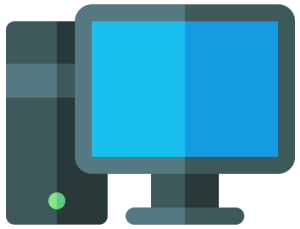
Port Numbers



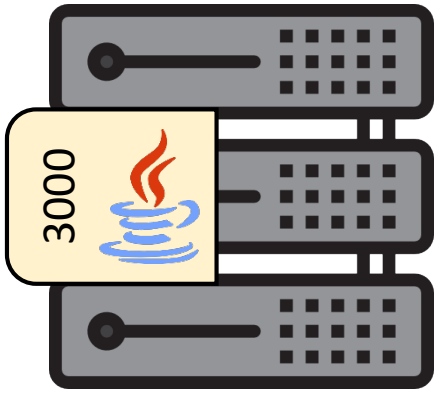
Client Server

Establish connection with IP address and port

Transfer data between the client and server



Client



Server

Either party can terminate the connection

TCP and UDP

TCP	UDP
Connected	Connectionless
State Memory	Stateless
Byte Stream	Packet/Datagram
Ordered Data Delivery	No Sequence Guarantee
Reliable	Lossy
Error Free	Error Packets Discarded
Handshake	No Handshake
Flow Control	No Flow Control
Relatively Slow	Relatively Fast
Point to Point	Supports Multicast
Security: SSL/TLS	Security: DTLS

Image from https://www.netburner.com/learn/tcp-vs-udp-battle-of-the-protocols/

Client/Server - Establish Connection

```
ServerSocket server = new ServerSocket(3000);
Socket socket = server.accept();
```

Opens a port to accept connection

Block until a connection arrives

Once connected, server socket will be connected to the client's socket endpoint

```
Socket socket = new Socket("locahost", 3000);
```

Once connected, client socket will be connected to the server's socket endpoint

Tries to connect to an application on localhost listening on port 3000

Client/Server - Establish Connection

```
try (InputStream is = socket.getInputStream()) {
    BufferedInputStream bis = new BufferedInputStream(is);
    DataInputStream dis = new DataInputStream(bis);
    String line = dis.readUTF();
    ...
}
```

Get the input stream from the server socket endpoint

Determine how the data should be read

Read the data. Block until data arrives

```
try (OutputStream os = socket.getOutputStream()) {
    BufferedOutputStream bos = new BufferedOutputStream(os);
    DataOutputStream dos = new DataOutputStream(bos);
    dos.writeUTF("hello, world");
    dos.flush();
    ...
}
```

Should match

Determine how the data should be transmitted

Write the data the server

Flush the output stream to send the data

Client/Server - Close Connection

```
try (InputStream is = socket.getInputStream()) {
    ...
    String line = dis.readUTF();
    ...
} catch (EOFException e) {
    socket.close();
}
```

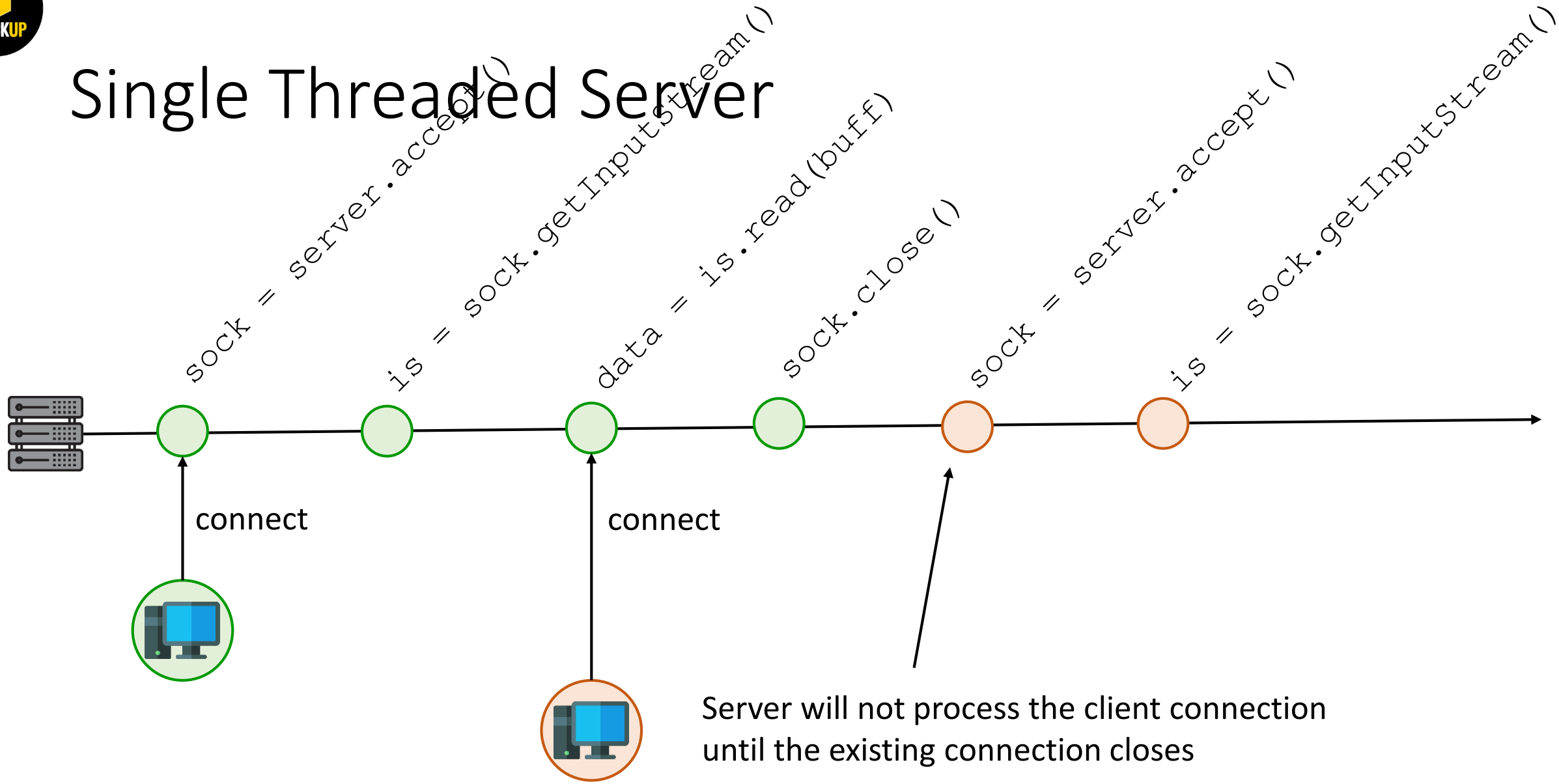
Throws an `EOFException` when socket is closed

Clean up by closing the socket

```
socket.close();
```

Client closes the connection. Note the server can close the connection as well

Single Threaded Server



Threads

- Thread is an execution path of a program
  - When Java application runs, the JVM creates and run the program with a main thread
- When the main thread is engaged in some activity, it will not be able to perform other tasks
- Threads are like mini programs running within the main program
  - Run independently of the man thread
  - Can communicate with the main thread with pipes, locks and semaphores
  - Need to prevent race condition where 2 or more thread concurrently update an object
- Note: will not cover multi-threaded programming

Creating a Thread

- Class must implement the `Runnable` interface
  - Has one method `run()` which is the thread's body
  - When exit `run()`, the thread dies

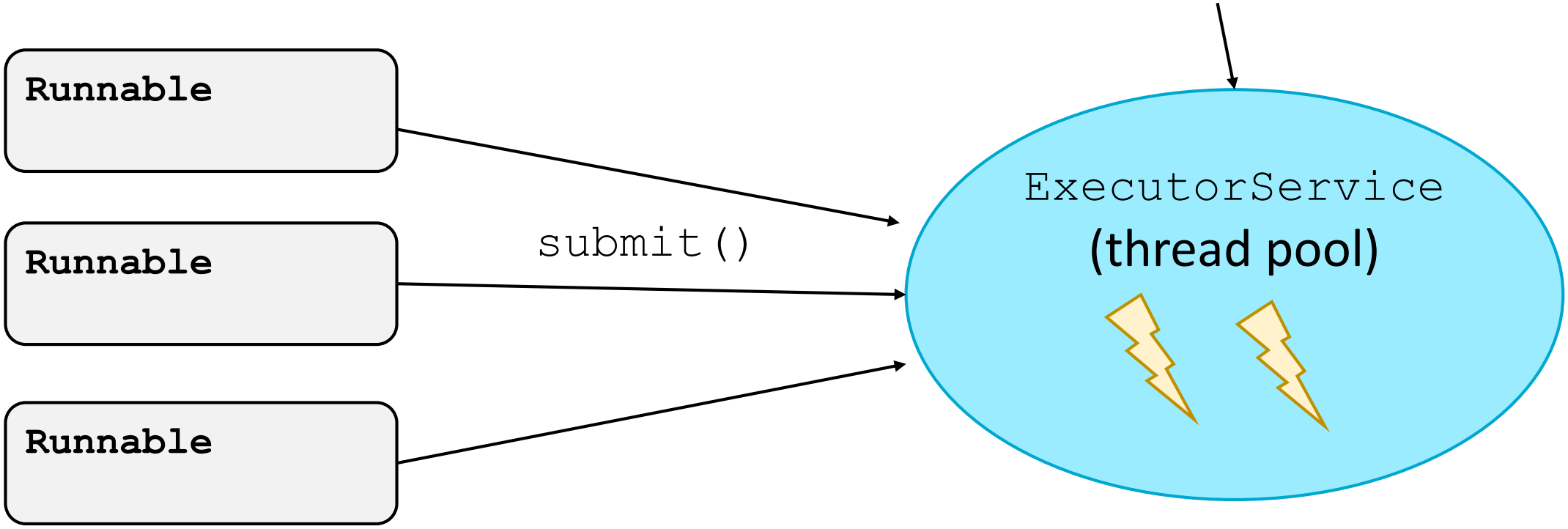
```
public class ConnectionHandler implements Runnable {
    private final Socket socket;
    public ConnectionHandler(Socket socket) { this.socket = socket; }

    @Override
    public void run() {
        try (InputStream is = socket.getInputStream()) {
            // as before
            ...
        }
    }
}
```

When the thread starts, it will call `run()`

Thread will stop when it exits `run()`

Executing a Thread

`ExecutorService` is thread pool contains a configurable number of threads



`Runnable`s will be scheduled by the `ExecutorService` to on the available thread.
`Runnable`s will have to wait if there are not enough free threads in the pool

Example - Multithreaded Server

Create a thread pool with 3 threads

```
ExecutorService threadPool = Executors.newFixedThreadPool(3);

ServerSocket server = new ServerSocket(port);

while (true) {
    Socket socket = server.accept();

    ConnectionHandler worker = new ConnectionHandler(socket);

    threadPool.submit(worker);
}
```

Instantiate a new `Runnable` `Runnable` should be self contained

Main thread returns to wait for new connection

Dispatch the `Runnable` to the thread pool. Will start executing if there are free thread

Full Duplex

- TCP is a full duplex protocol
  - Data can be read and written by both client and server at the same time
- Input stream is blocking, viz. will wait until data is available or socket connection is closed
  - Read and write must follow a certain sequence
  - Cannot receive data in real time
- Most efficient way to handle this is to use non-blocking I/O
  - Can use thread but not that efficient because every worker thread need an extra thread to block on read
- No blocking I/O will not be covered in this course